VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

VIZUALIZACE ALGORITMŮ PRO PLÁNOVÁNÍ CESTY

VISUALISATION OF PATH PLANNING ALGORITHMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **ŠIMON GALBA**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. JAROSLAV ROZMAN, Ph.D.**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Galba Šimon**
Program: Informační technologie
Název: **Vizualizace algoritmů pro plánování cesty**
Visualisation of Path Planning Algorithms
Kategorie: Umělá inteligence

Zadání:

1. Nastudujte algoritmy pro plánování cesty používané v robotice (bug algoritmy, potenciálové funkce a buněčné dekompozice a další). Nastudujte bakalářskou práci Jakuba Rusnáka "Vizualizace algoritmů pro plánování cesty".
2. Nad rámec práce J. Rusnáka navrhněte sadu dalších zásuvných modulů s vybranými algoritmy, kterými výsledný program rozšíříte.
3. Implementujte navržené zásuvné moduly a ke každému algoritmu vytvořte přehledný popis teorie nutné k pochopení jeho funkce.
4. Výslednou práci zhodnoťte a navrhněte vylepšení.

Literatura:

- Howie Choset et al., Principles of Robot Motion, 2005, ISN 0-262-03327-5.
- Rusnák Jakub, Vizualizace algoritmů pro plánování cesty, bakalářská práce, FIT VUT v Brně, 2017.

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Rozman Jaroslav, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 11. listopadu 2020

Abstrakt

Seminárna práca sa venuje štúdiu problematiky vizualizácie plánovačov cesty, popisu teórie konkrétnych algoritmov použitých na tieto účely a analýze už naimplementovaných algoritmov v diplomovej práci na ktorú táto seminárna práca nadväzuje.

Abstract

This term project attends to studying of path planning algorithm visualisation, theoretical description of algorithms used for this purpose and analysis of already developed algorithms in the master thesis that this term thesis builds upon.

Kľúčové slová

java, plánovanie cesty, analýza, vizualizácia

Keywords

java path planning, analysis, visualisation

Citácia

GALBA, Šimon. *Vizualizace algoritmů pro plánování cesty.* Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jaroslav Rozman, Ph.D.

Vizualizace algoritmů pro plánování cesty

Prehlásenie

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jaroslava Rozmana, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Šimon Galba
12. mája 2021

Poďakovanie

Rád by som poďakoval vedúcemu práce Ing. Jaroslavovi Rozmanovi za cenné rady a vedenie pri tvorbe práce. Taktiež by som rád poďakoval rodine za podporu pri celom štúdiu a mojej priateľke za trpezlivosť a vytvorenie prostredia na tvorbu.

Obsah

Kapitola 1

Úvod

Pri štúdiu informačných technológií je pochopenie funkčnosti algoritmov často zdĺhavým a nepríjemných procesom. Z vlastných skúseností a pozorovania okolia som zistil, že nepohodlne veľká časť študentov sa „memoruje“ a od slova do slova učí pseudokódy algoritmov pred záverečnými skúškami predmetov bez toho aby mali všeobecné povedomie o ich podstate a fungovaní. Je ťažké z tejto skutočnosti obviňovať len študentov. Pri vlastných štúdiách som často narazil na problém pochopenia algoritmu z čistého textu. Mnohokrát trvalo bolo potrebné mnohohodinové kreslenie, vysvetľovanie či dokonca predstavovanie na pochopenie funkčnosti daného algoritmu. Priemerný študent teda radšej zvolí cestu memorovania desiatok riadkov kódu na prejdenie skúšky s vedomím, že si môže daný algoritmus vždy „vygoogliť“. Spomínané kreslenie či predstavovanie je forma vizualizácie. Vizualizácia často pre študenta predstavuje hranicu medzi zmätením a pochopením. Spojenie vizualizácie s teoretickými znalosťami danej látky, v tomto prípade algoritmu, môže viesť k ušetreniu vzácneho času študenta pri vzdelávaní. V tejto práci sa budem venovať konkrétne vizualizácii algoritmov pre plánovanie cesty. Budem nadväzovať na diplomovú prácu Jakuba Rusnáka a využívať ním vytvorenú knižnicu a aplikáciu na vizualizáciu algoritmov. Technickou časťou tejto práce bude práve vytvorenie zásuvných modulov poskytujúcich možnosť zobrazenia viacerých algoritmov pre rozšírenie možností aplikácie. V teoretickej časti práce budú tieto algoritmy popísané odborne a takisto budú zobrazené ich pseudokódy.

Kapitola 2

Algoritmy plánovania cesty

2.1 Plánovače cesty a konfiguračný priestor

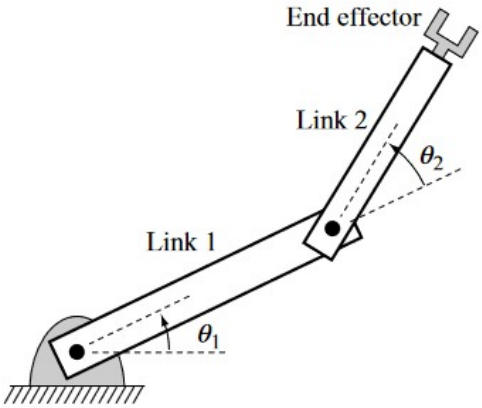
Pred začatím plánovania cesty pre robotov si najprv musíme uviesť potrebné obmedzenia a spôsoby takéhoto plánovania. Základnou podmienkou, aby sme boli schopný plánovať cestu nejakého robota je vedieť jeho pozíciu v každom momente cesty. Nestačí nám však vedieť polohu jeho stredu, pretože pri počítačovej simulácii je potrebné zaručiť, že sa v žiadnom okamihu simulácie žiadna časť tela robota nenachádza v inom objekte (koliduje). Riešenie tejto problematiky je popísané v kapitole 4.2.

Pre presné špecifikovanie polohy robota bol zavedený pojem **konfigurácia** a **konfiguračný priestor**.

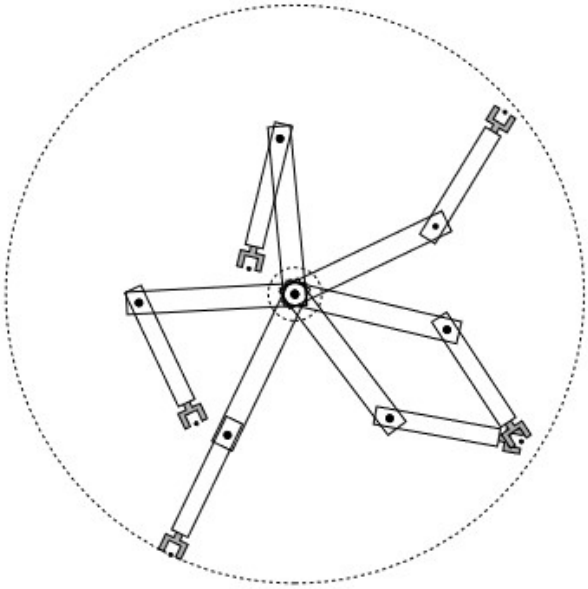
- **Konfigurácia** robota je kompletná špecifikácia pozície každého bodu ktorý objekt robot obsahuje [3] [4]
- **Konfiguračný priestor** robota je priestor všetkých možných konfigurácií systému či objektu robota [3] [4]

Z týchto definícia vychádza, že konfigurácia je jednoducho povedané bod v abstraktnom konfiguračnom priestore. Pre znázornenie týchto definícií si môžeme robota predstaviť ako mobilného robota v priestore, ktorý sa dokáže pohybovať v každom smere a nemusí pritom rotovať. Potom jednoduchá reprezentácia konfigurácie robota je poloha jeho stredu (x, y), voči fixnému bodu v priestore.

Pre predstavenie si môžeme zadefinovať robotické dvoj-kĺbové rameno z obrázku 2.1 s jedným bodom zakoreneným v určitom bode priestoru a jediný pohyb ramena je teda rotácia okolo tohto bodu, a bod druhej časti ramena je pripojený na koniec prvej časti tak, že jediný pohyb druhej časti je rotácia okolo tohto bodu. Jedna konfigurácie je teda nehybné rameno, v tomto prípade definované uhlami medzi jednotlivými časťami ramena 2.1a a konfiguračný priestor je množina všetkým kombinácií uhlov ramena a teda pozície konca ramena v priestore 2.1b.



(a) Konfigruácia ramena, definovaná uhlami θ_1 a θ_2



(b) Konfiguračný priestor ramena, všetky možnosti a kombinácie uhlov θ_1 a θ_2

Obr. 2.1: Znázornenie konfigurácie (a) a konfiguračného priestoru (b) dvoj kĺbového robotického ramena. Prevzaté z [3]

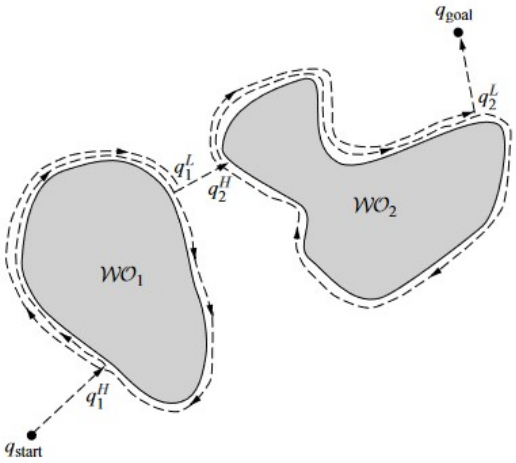
2.2 Bug algoritmy

Vhodnosť bug algoritmov pre študijné a demonštračné účely spočíva okrem iného v ich matematickej nenáročnosti. Bug 1 a Bug 2 algoritmy sú pomerne jednoduché na pochopenie a vizualizáciu. Primárnym praktickým účelom pri používaní bug algoritmov je navigácia bodového robota zo známeho počiatku do známeho cieľa cez 2D prostredie obsahujúce neznáme prekážky. Uvažujeme, že robot disponuje buď dotykovým, alebo vzdialenostným senzorom, ktorý dokáže rozpoznať stret s prekážkou. Taktiež predpokladáme, že robot dokáže zaznamenávať svoju relatívnu polohu v prostredí. V prípade, že je robot vybavený

nenulovým vzdialenostným senzorom, môžeme použiť Tangent Bug algoritmus na nájdenie rýchlejšej cesty do cieľa. Pohyby robota ovládaného pomocou bug algoritmov sú taktiež jednoduché. Robot sa bude pohybovať rovno po priamej čiare smerom k cieľu, alebo bude kopírovať hranu prekážky. Medzi týmito dvomi stavmi bude robot prepínať na základe údajov z jeho dotykového, či vzdialenostného senzora. Vďaka týmto dvom vlastnostiam vieme zaručiť, že BUG 1 aj BUG 2 algoritmus sú kompletné.

2.2.1 Bug 1

Bug I algoritmus využíva veľmi prirodzenú techniku hľadania cesty do cieľa. Nasleduje priamu čiaru z počiatočného bodu do koncového, pokiaľ nenarazí na prekážku. Tu si musíme určiť ďalšiu predpokladanú vlastnosť robota, a tou je možnosť určiť a zapamätať si vzdialenosť dvoch bodov. Pri narazení na prekážku si robot ovládaný pomocou Bug 1 algoritmu zapamätá bod stretu s prekážkou, nazvime ho Q_1^H. Robot potom obíde celú prekážku, pričom hľadá bod na obvode prekážky, ktorý je najbližšie cieľovému bodu. Keď robot opäť dosiahne bod Q_1^H, vydá sa do bodu ktorý vyhodnotil ako najbližší k cieľu. Tento bod nazývame bod opustenia prekážky a označíme ho Q_1^L. Odtiaľ znovu nasleduje priamku smerom k cieľu. V prípade, že opätovne narazí na rovnakú prekážku (tento náraz by sa stal v tom istom okamihu ako opustenie hrany prekážky), vyhodnotí, že nevie nájsť cestu k cieľu v danom priestore. V opačnom prípade pokračuje po priamke a prípadne daný proces opakuje pre iné prekážky.



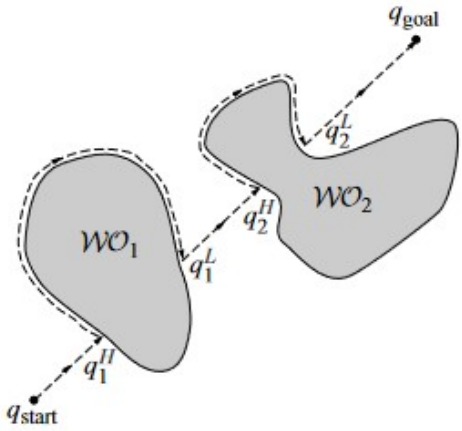
Obr. 2.2: Nájdenie cesty konfiguračným priestorom pomocou algoritmu Bug1 [3]

Algorithm 1 Bug 1

while *True* **do**
  while q_{goal} *nie je nájdený alebo nie je nájdená prekážka v bode* q_i^H. **do**
    Z bodu q_{i-1}^L *sa pohybuj smerom k* q_{goal}.
  if q_{goal} *je dosiahnutý* **then**
    Exit
  while q_{goal} *nie je dosiahnutý a prekážka nie je znovu stretnutá* **do**
    Nasleduj obrys prekážky.
  Urči bod q_i^L *na obryse prekážky ktorý je najbližšie k bodu* q_{goal}.
  Presuň sa na bod q_i^L.
  if *robot sa pri presúvaní pohybuje smerom k cieľu* **then**
    Zhodnoť, že q_{goal} *nie je dosiahnuteľný.*
    Exit

2.2.2 Bug 2

Pri alternatíve Bug II predpokladáme rovnaké schopnosti robota ako pri algoritme Bug I. Na rozdiel od algoritmu Bug I bude Bug II pracovať len s jednou priamkou, označme ju m_1, ktorá vedie z počiatočného bodu do koncového. Robot sa vydá po tejto priamke a pokračuje až pokiaľ nepríde do cieľa alebo nenarazí na prekážku v priestore. V prípade, že narazí na prekážku, začne kopírovať jej hranu rovnako ako pri algoritme Bug I, ale robí tak len do bodu, kým opätovne nepretne pomyselnú priamku m_1. V tomto prípade sa znovu vydá po spomínanej priamke smerom k cieľu.



Obr. 2.3: Nájdenie cesty konfiguračným priestorom pomocou algoritmu Bug2 [3]

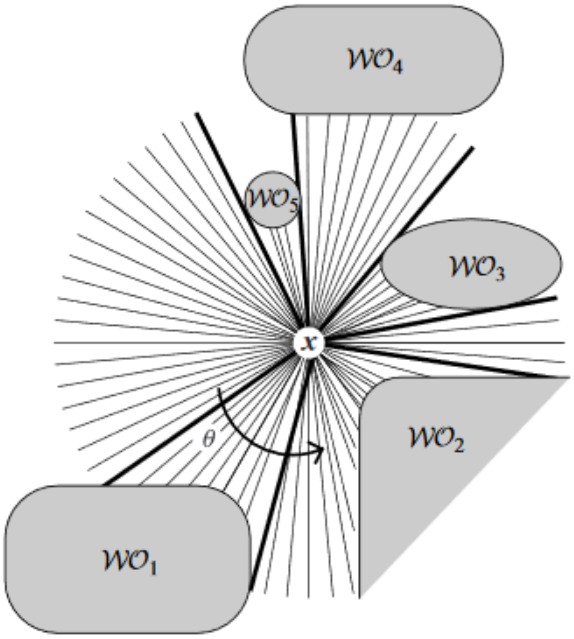
Algorithm 2 Bug 2

1: **while** *True* **do**
2: **while** q_{goal} *nie je nájdený alebo nie je nájdená prekážka v bode* q_i^H. **do**
3: *Z bodu* q_{i-1}^L *sa pohybuj smerom k* q_{goal} *po priamke* m_1.
4: **if** q_{goal} *je dosiahnutý* **then**
5: *Exit*
6: **while** q_{goal} *nie je dosiahnutý*
& prekážka nie je znovu nájdená
& priamka m-line nie je dosiahnutá v bode Q takom, že $Q_6 = H_i$, d(Q, goal) < d(H_i, goal), a priamka (Q, goal) nepretína prekážku v bode Q **do**
7: *nech* $q_{i+1}^L = m$
8: *inkrementuj i*

Pri jednoduchých konvexných prekážkach je algoritmus Bug 2 bezpochyby rýchlejší ako Bug I, lenže tento pomer sa môže meniť v závislosti na tvare a počte prekážok v priestore. Z textového popisu môže byť ťažké predstaviť si tento rozdiel, avšak vizualizácia môže pomôcť tento fakt ozrejmiť aj menej predstavivým, či skúseným študentom/záujemcom.

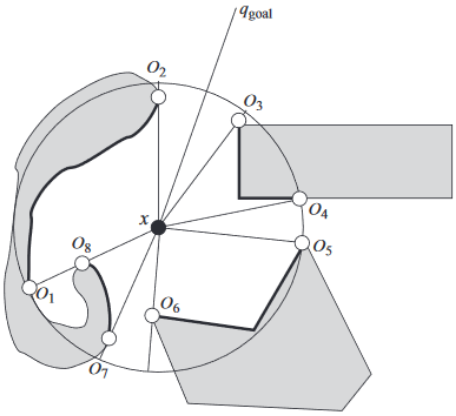
2.2.3 Tangent bug

Ak je robot vybavený už spomínaných vzdialenostným senzorom, môžeme optimalizovať algoritmus Bug 2 pomocou algoritmu Tangent Bug. Vďaka vzdialenostnému senzoru je robot schopný rozpoznávať ne-spojitosti vo vysielaných lúčoch signálu. V prípade narušenia lúča signálu robot zaznamenáva prekážku. Na ilustračnom obrázku 2.4 sú nájdené ne-spojitosti zobrazené ako hrubšie lúče signálu. Tenšie lúče sú tie, ktoré nenarazili na žiadnu prekážku a pre nášho robota naberajú efektívne nekonečnú veľkosť.



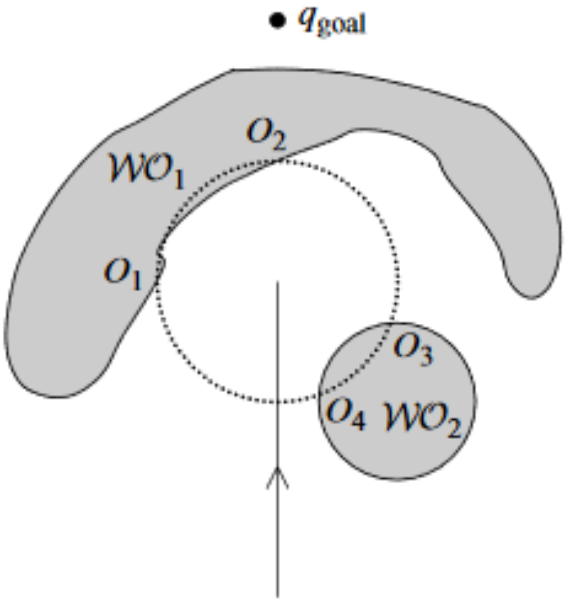
Obr. 2.4: Obrázok zobrazuje nájdené ne-spojitosti hrubými lúčmi [3]

Vďaka týmto ne-spojitostiam si dokážeme v grafe vyznačiť a zapamätať body korešpondujúce s hranami prekážok. V prípade, že senzor má fixný (konečný) maximálny dosah a táto vzdialenosť nie je dostatočná na odhalenie rohu prekážky, pridáme bod na hrane prekážky v maximálnej vzdialenosti senzora (bod O_4 v obrázku 2.5).



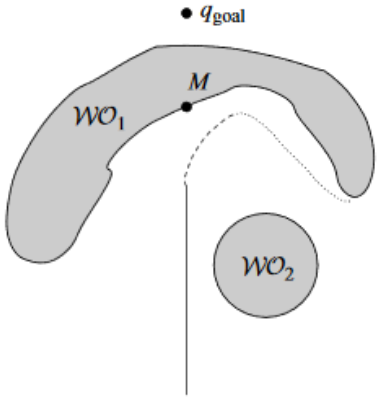
Obr. 2.5: Hrany prekážok a body s končiacim dosahom senzoru [3]

Robot sa opäť pohybuje na priamke smerom k cieľu, no pri tangent algoritme sa tento pohyb mení už keď robot odhalí prekážku ktorá leží na priamke smerom k cieľu. V prípade, že robot odhalí takúto prekážku, označuje si nové body ležiace na priesečníkoch rádiusu senzora a hrany prekážky 2.6.

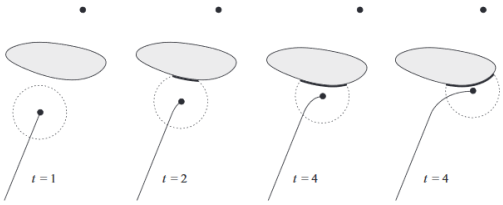


Obr. 2.6: Zaregistrovanie prekážky, ktorá stojí medzi robotom a cieľom [3]

Potom mení svoj smer k bodu, ktorý najviac heuristicky znižuje vzdialenosť k cieľu. Ako sa robot približuje k tomuto bodu, a teda k prekážke, súradnice priesečníkov sa posúvajú a teda sa mení aj smer pohybu robota (obrázok 2.7 + obrázok 2.8)



Obr. 2.7: Efektívne vyhnutie sa prekážke s meniacim sa bodom priesečníku rádiusu senzora [3]



Obr. 2.8: Vizualizácia krokov pri obchádzaní inej prekážky ležiacej medzi robotom a cieľom [3]

Algorithm 3 Tangent Bug [3]

Input:

Robot so vzdialenostným senzorom

Output:

Cesta do konfgurácie q_{goal} alebo oznámenie o nedostupnosti

1: **while** True **do**
2: **repeat**
3: Pohyb robota smerom k bodu $n \in T, O_i$ ktorý minimalizuje $d(x, n) + d(n, q_{goal})$
4: **until**

- q je bez kolízií
- smer pohybu robota minimalizujúci $d(x, n) + d(n, q_{goal})$ začne zväčšovať $d(x, q_{goal})$, teda nájde lokálne minimum

5: Vyber smer obchádzania prekážky nadväzujúci na smer najposlednejšieho pohybu
6: **repeat**
7: aktualizuj hodnoty d_{reach}, $d_{followed}$ a $\{O_i\}$.
8: Pohybuj sa smerom k bodu $n \in \{O_i\}$ ktorý je v smere vybranej hranice prekážky
9: **until**

- robot nenájde cieľ
- robot nedokončí okruh okolo celej prekážky a teda vie, že je cieľ nedostupný
- $d_{reach} < d_{followed}$

2.3 Pravdepodobnostné algoritmy

Súčasťou tejto práce bolo naštudovať, testovať a prípadne opraviť pravdepodobnostné algoritmy vytvorené Jakubom Rusnákom v jeho diplomovej práci.
V nasledujúcich podkapitolách budú pravdepodobnostné algoritmy teoreticky popísané a v kapitole 5.1.1 je uvedený postup a výsledky ich testovania.

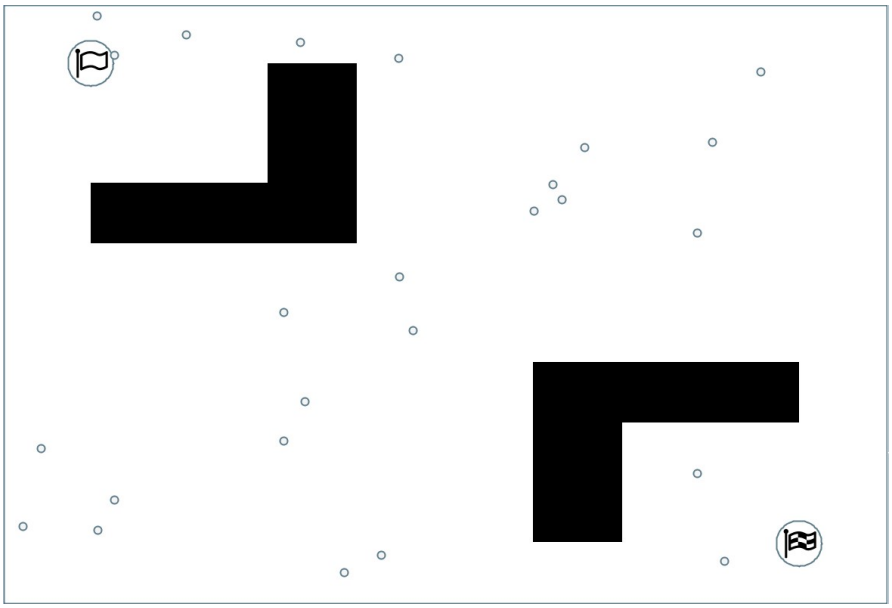
2.3.1 Pravdepodobnostná roadmapa

Algoritmus pravdepodobnostnej roadmapy [PRM] sa skladá z dvoch hlavných fáz, fáza konštrukcie a vyhľadávacia fáza. Výsledkom algoritmu je bezkolízna trasa konfiguračným priestorom alebo informácia a nedostupnosti cieľa.

Konštrukčná fáza

Na začiatku konštrukcie je potrebné zadefinovať neorientovaný graf G s vrcholmi V a hranami E tak, že $G = (V, E)$. Pravdepodobnostná roadmapa sa potom po konštrukcii bude ukladať to tohto grafu G.

Konštrukcia prebieha inkrementačne s pravdepodobnostnou charakteristikou. Opakovane sa do množiny vrcholov V pridáva náhodne vygenerovaná konfigurácia c za podmienky, že c patrí do konfiguračného priestoru čo prakticky znamená, že neleží v žiadnej prekážke 2.9. Po pridaní vyžadovaného počtu konfigurácií sa opakovane vytvárajú hrany grafu a pridávajú do množiny E nasledujúcim spôsobom:
Pre každý vrchol v z množiny V sa vyberie počet n najbližších vrcholov z množiny V, kde počet n je vstupným parametrom algoritmu. Pre každú dvojicu v a každý z n najbližších vrcholov sa vytvorí hrana a ak sa ešte nenachádza v množine E, vloží sa 2.10.
Takto vzniká neorientovaný graf G, bez záruky spojitosti, ktorá sa pri jednoduchom algoritme [PRM] v tejto fáze nekontroluje. [3][2]



Obr. 2.9: Konštrukčná fáza algoritmu PRM - vyvorenie a vloženie vrcholov do konfiguračného priestoru [5]

Algorithm 4 Pseudokód konštrukčnej fázy PRM [3]

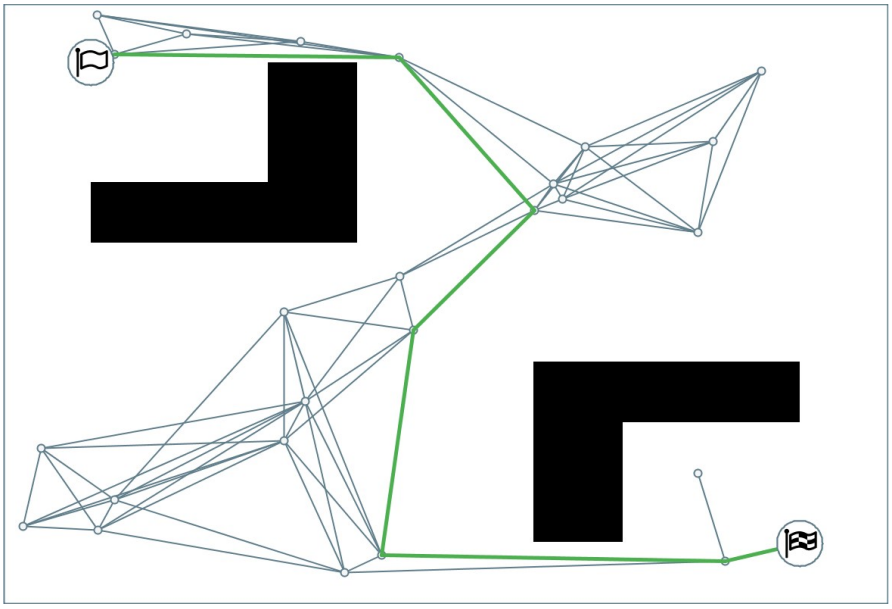
Input:
n : počet vrcholov na pridanie do roadmapy
k : počet najbližších susedov na vytvorenie hrán
Output:
Roadmapa $G = (V, E)$

```
1:  V ← ∅
2:  E ← ∅
3:  while |V| < n do
4:      repeat
5:          q ← náhodná konfigurácia
6:      until q je bez kolízií
7:      V ← V ∪ {q}
8:  for all q ∈ V do
9:      N_q ← k najlbližších susedov q vybraných z V
10:     for all q' ∈ N_q do
11:         if (q, q') ∉ E ∨ Δ(q, q') ≠ NIL then
12:             E ← E ∪ {(q, q')}
```

Vyhľadávacia fáza

Cieľom vyhľadávacej fázy je nájsť cestu medzi počiatočnou konfiguráciou q_{init} a koncovou konfiguráciou q_{goal}, ak existuje. S grafom roadmapy G vytvorenom v konštrukčnej fáze sa pre každý z k najbližších vrcholov v z množiny V vytvorí nová hrana e a v prípade, že sa nenachádza v prekážke sa pridá do grafu roadmapy. Ak sa konfigurácia q_{init} alebo q_{goal} nemôže pripojiť do grafu, algoritmus oznamuje nemožnosť nájdenia cesty.

Ak sa obidve konfigurácie do grafu pridajú, využije sa vyhľadávacia heuristika (napríklad Dijkstrov algoritmus, A*). Podľa zvolenej heuristiky potom dostávame cestu grafom spájajúcu konfigurácie q_{init} a q_{goal}, prípadne algoritmus oznamuje neúspech.



Obr. 2.10: Vyhľadávacia fáza algoritmu PRM - pripojenie počiatočnej a cieľovej konfigurácie do grafu a nájdenie cesty [5]

Algorithm 5 Pseudokód vyhľadávacej fázy PRM. [3]

Input:

q_{init}: počiatočná konfigurácia
q_{goal}: cieľová konfigurácia
k: počet najbližších susedov
$G = (V, E)$: roadmapa vytvorená konštrukčnou fázou PRM

Output:

cesta z q_{init} do q_{goal} alebo oznámenie nenájdenia cesty

$N_{q_{init}} \leftarrow k$ najbližších susedov uzla q_{init} z V
$N_{q_{goal}} \leftarrow k$ najbližších susedov uzla q_{goal} z V
$V \leftarrow \{q_{init}\} \cup \{q_{goal}\} \cup V$
$q' \leftarrow$ najbližší sused uzla q_{init} z $N_{q_{init}}$
repeat
    if $\Delta(q_{init}, q') \neq NIL$ **then**
        $E \leftarrow E \cup (q_{init}, q')$
    else
        $q' \leftarrow$ nasledujúci sused uzla q_{init} z $N_{q_{init}}$
until spojenie uzlov bolo úspešné alebo je $N_{q_{init}}$ prázdny
$q' \leftarrow$ najbližší sused uzla q_{goal} z $N_{q_{goal}}$
repeat
    if $\Delta(q_{goal}, q') \neq NIL$ **then**
        $E \leftarrow (q_{goal}, q' \cup E)$
    else
        $q' \leftarrow$ nasledujúci sused uzla q_{goal} z $N_{q_{goal}}$
until spojenie uzlov bolo úspešné alebo je $N_{q_{goal}}$ prázdny
$P \leftarrow$ najkratšia cesta z q_{init} do q_{goal} v G
if P nie je prázdny **then**
    return P
else
    return neúspech

2.3.2 Expanzívne stromy

Metóda expanzívnych stromov [EST] bola pôvodne navrhnutá ako efektívny jedno-dotazový plánovač na pokrytie priestoru medzi konfiguráciami q_{init} a q_{goal}. [3]

Konštrukcia stromov

Nech je T jeden zo stromov T_{init} zakorenený v q_{init} alebo T_{goal} zakorenený v q_{goal}. Plánovač najprv vyberie konfiguráciu q v strome T z ktorej bude rásť strom T a potom vyberie náhodnú konfiguráciu q_{rand} z uniformného rozloženia v okolí konfigurácie q, pričom konfigurácia q je vybraná náhodne s pravdepodobnosťou $\pi_T(q)$. Lokálny plánovač δ sa pokúsi spojiť konfigurácie q a q_{rand}. Pri úspešnom spojení je q_{rand} pridaná do množiny vrcholov stromu T a hrana (q, q_{rand}) do množiny hrán. Tento proces sa opakuje kým nie je vložený určený počet konfigurácií 2.11.

Algorithm 6 Pseudokód zostavenia stromu algoritmom EST. [3]

Input:

q_0: konfigurácia v ktorej je strom zakorenený

n: počet pokusov na expandovanie stromu

Output:

strom $T = (V, E)$ zakorenený v q_0 ktorý obsahuje $\leq n$ konfigurácií

1: $V \leftarrow q_0$
2: $E \leftarrow \emptyset$
3: **for** $i = 1$ do n **do**
4: $q \leftarrow$ konfigurácia náhodne vybraná z T s pravdepodobnosťou $\pi_T(q)$
5: extend EST(T, q)
6: **return** T

Počas konštrukcie stromu sa volá funkcia expanzie popísaná nasledujúcim pseudokódom.

Algorithm 7 Pseudokód expanzie stromu algoritmom EST. [3]

Input:

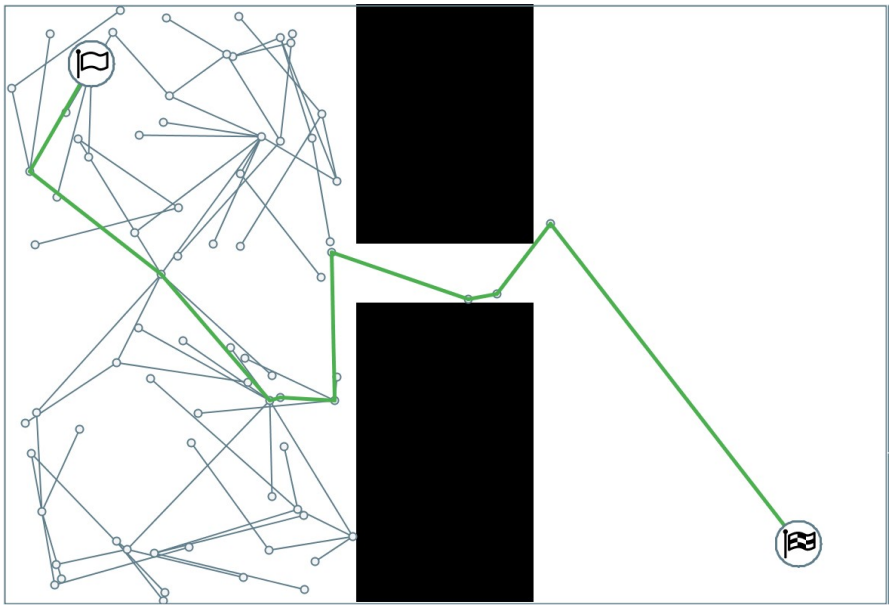
$T = (V, E)$: expanzívny strom

q: koreňová konfigurácia stromu T

Output:

nová konfigurácia q_{new} v okolí q, alebo `NIL` v prípade neúspechu

1: $q_{new} \leftarrow$ náhodne vybraná konfigurácia z okolia q bez kolízií s prekážkami
2: **if** $\Delta(q, q_{new})$ **then**
3: $V \leftarrow V \cup \{q_{new}\}$
4: $E \leftarrow E \cup \{(q, q_{new})\}$
5: **return** q_{new}
6: **return** `NIL`



Obr. 2.11: Algoritmus EST - náhodne vybrané konfigurácie sa expandujú smerom od štartu [5]

2.3.3 Rapídne sa rozširujúce stromy

Metóda rapídne sa rozšírujúcich stromov [RRT] bola pôvodne navrhnutá ako efektívna jedno-dotazový plánovač na pokrytie priestoru medzi konfiguráciami q_{init} a q_{goal}. [3]

Konštrukcia RRT

Nech je T jeden zo stromov T_{init} zakorenený v q_{init} alebo T_{goal} zakorenený v q_{goal}. Každý strom sa inkrementačne rozširuje. S každou iteráciou sa náhodne vygeneruje konfigurácia q_{rand} a pokúsi sa spojiť s najbližšou existujúcou konfiguráciou v strome T úsekmi o vzdialenosti $step_size$ definovanej pri spustení algoritmu. Ak sa žiadny z týchto úsekov nenachádza v prekážke, úseky sa pridajú do grafu.

Algorithm 8 Pseudokód konštrukcie stromu algoritmom RRT. [3]

Input:
q_0: koreňová konfigurácia stromu
n: počet pokusov na expandovanie stromu
Output:
strom $T = (V, E)$ zakorenený v q_0 ktorý obsahuje $\leq n$ konfigurácií
1: $V \leftarrow \{q_0\}$
2: $E \leftarrow \emptyset$
3: **for** $i = 1$ do n **do**
4: $q_{rand} \leftarrow$ náhodná voľná konfigurácia
5: extend RRT(T, q_{rand})
6: **return** T

Algorithm 9 Pseudokód expnazie stromu algoritmom RRT. [3]

Input:
$T = (V, E)$: strom RRT
q: konfigurácia ku ktorej sa expanduje strom T
Output:
nová konfigurácia q_{new} smerujúca ku q, alebo `NIL` v prípade neúspechu algoritmu
1: $q_{near} \leftarrow$ najbližší sused q zo stromu T
2: $q_{new} \leftarrow$ postup po hrane q_{near}, q_{rand} vo vzdialenosti `step_size` od q_{near}
3: **if** q_{new} nie je v kolízii s prekážkou **then**
4: $V \leftarrow V \cup \{q_{new}\}$
5: $E \leftarrow E \cup \{(q_{near}, q_{new})\}$
6: **return** q_{new}
7: **return** `NIL`

Jedna z možných úprav algoritmu je dynamické prideľovanie hodnoty premennej *step size* podľa vzdialenosti konfigurácií q_{near} a q_{new}. Čím väčšia vzdialenosť, tým väčší krok.

Algorithm 10 Pseudokód spájania grafu algoritmu RRT. [3]

Input:
$T = (V, E)$: strom RRT
q: konfigurácia ku ktorej sa expanduje strom T
Output:
`connected` ak sa q pripojí; inak `failure`
1: **repeat**
2: $q_{new} \leftarrow$ extend RRT(T, q)
3: **until** ($q_{new} = q$ **or** $q_{new} = NIL$)
4: **if** $q_{new} = q$ **then**
5: **return** `connected`
6: **else**
7: **return** `failure`

2.4 Ostatné algoritmy

V tejto kapitola sa nachádza teoretický úvod do algoritmov, ktoré som v rámci práce študoval a vytváral tento popis. V implementačnej časti som sa však rozhodol implementovať Bug algoritmy a venovať sa zmene architektúry aplikácie. Opis týchto algoritmov tu však pre úplnosť ponechám.

Potenciálové funkcie

V tejto kapitole sa pozrieme na alternatívnu reprezentáciu priestoru v ktorom sa náš robot pohybuje. Pri využití potenciálových funkcií definujeme metódy ktoré majú širšie možnosti využitia vďaka všeobecnejšej definícii a matematickej notácie.
Využitie potenciálovej funkcie nám umožní "tlačiť"na robota ako na časticu v potenciálovom vektorovom poli s gradačnou silou. Intuitívne by sme mohli túto vlasnosť vysvetliť tak, že pozitívne nabitú časticu (robota) priťahuje negatívne nabitý bod (cieľ). Ak je teda celý priestor vyplnený takýmito vektormi, robot vždy nájde cestu k svojmu cieľu. Mohlo by sa

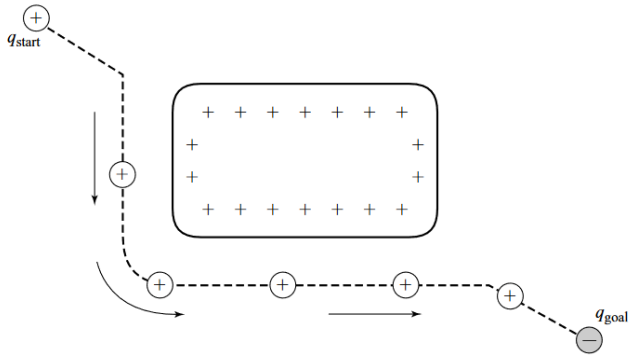
samozrejme stať, že vektor bude "tlačiť"robota do prekážky a ten sa nebude pohybovať ďalej. Preto budú všetky prekážky rovnako pozitívne nabité a teda budú nášho robota odpudzovať. Kombinácia odpudzovania od prekážok a priťahovania smerom k cieľu by mala zabezpečiť nájdenie cesty medzi počiatočným bodom a cieľom. Potenciálovú funkcie môžeme zaznačiť ako:

$$U : \mathbb{R}^m -> \mathbb{R}$$

Vektor tejto funkcie označujúci silu v danom bode, ukazujúci smer ktorý lokálne maximalizuje U nazývame Gradient a použijeme ho na definovanie vektorového poľa ktoré priradí vektor každému bodu nášho priestoru. Matematicky ho môžeme zaznačiť následovne:

$$\nabla U(q) = DU(q)^T = [\frac{\partial U}{\partial q_1}, ..., \frac{\partial U}{\partial q_m}(q)]^T$$

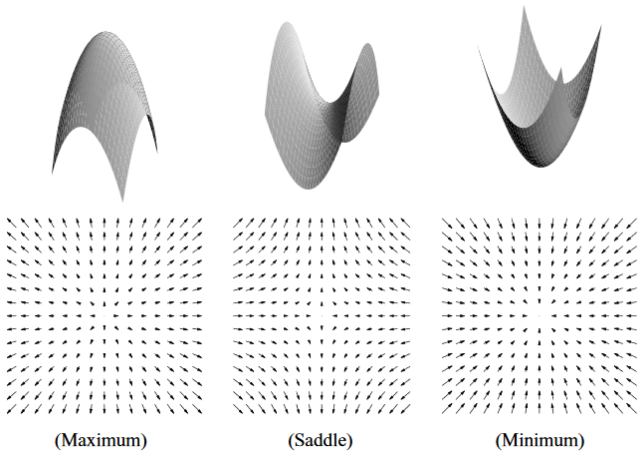
Vizuálne znázornenie predošlého textu. Pozitívne nabitá prekážka odpudzuje robota na ceste k negatívne nabitému cieľu a robot efektívne túto prekážku obchádza:



Obr. 2.12: Kladne nabitý robot obchádza kladne nabitú prekážku a je priťahovaný záporne nabitým cieľom [3]

Ukončenie pohybu robota

Robot ukončí svoj pohyb nie keď nájde cieľ, ale keď sa dostane na miesto kde na neho nebude pôsobiť žiadny Gradient, teda v bode **q** takom, že $\nabla U(q) = 0$. Bod q môžeme potom nazvať *kritickým bodom* priestoru U. Reálne môže tento kritický bod označovať 3 stavy. Bod q je maximum, minimum alebo sedlový bod. Tieto tri typy sú znázornené na nasledujúcom obrázku:



Obr. 2.13: Zobrazenie stavu Gradientu v (z ľava) lokálnom maxime, sedlovom bode, lokálom minime [3]

Pre použitie potenciálovej funkcie pre pohyb robota v poli nie je potrebné brať do úvahy silu vektoru. Namiesto toho nám každý vektor označuje len smer pohybu v poli. Vďaka týmto vlastnostiam vieme vylúčiť, že sa bude robot nachádzať v lokálnom maxime pokiaľ tam robot nezačína. V takom prípade ho ale oslobodí ľubovoľný počiatočný pohyb a tento stav je ľahko ošetriteľný konkrétnym algoritmom. Pravdepodobnosť, že sa robot ocitne v sedlovom bode je pomerne nízka, no nie je nulová. Z tvaru vektorov okolo tohto bodu vyplýva, že nebude koncovým bodom grafu a teda je pre nás tento stav nežiadúci. Tento bod je avšak nestabilný, a pred ukončením algoritmu môžeme jednoducho overiť, či sa robot nahádza v lokálnom minime alebo v sedlovom bode. Stačí malý pohyb robota a zo sedlového bodu sa pomocou von smerujúcich vektorov dostane sám. Avšak ak sa robot nachádza v lokálnom minime, každý okolitý vektor bude pôsobiť presne opačným smerom ako je pohyb robota snažiaceho sa overením algoritmom z tohoto bodu dostať. Lokálne minimum budeme spravidla považovať za koncový stav.

Aditívny atraktívno-odpudivý potenciál

Najjednoduchšia potenciálna funkcia je práve atraktívno-odpudivý potenciál. Ako som už v opisoval, jeho použitie je priamočiare. Cieľ robota priťahuje a prekážky ho odpudzujú. V tejto kapitole si presnejšie rozoberieme princíp za touto metódou.
Najprv si zadefinujeme vzťah potenciálovej funkcie pre tento typ. Potenciálová funkcia môže byť v tomto prípade zostrojená ako sum príťažlivých a odpudivých síl:

$$U(q) = U_{att}(q) + U_{rep}(q).$$

Príťažlivý potenciál

Potenciálové pole U_{att} musí splňovať niekoľko podmienok. Okrem iného by malo byť monotónne rastúce priamo úmerne so zväčšujúcou sa vzdialenosťou od cieľa. Túto vlasnosť môžeme matematicky zapísať následovne:

$$U(q) = \zeta d(q, q_{goal})$$

Kde Zeta ζ značí koeficient váhy vzdialenosti použitý na škálovanie príťažlivého potenciálu. Rovnako si môžeme zaznačiť aj príťažlivý Gradient, ktorého vektor smeruje v každom bode

smerom od cieľa, a jeho nasledovanie v opačnom smere dovedie robota (v priestore bez prekážok) do cieľa:

$$\nabla U\ (q) = \frac{\zeta}{d(q, q_{goal}}(q - q_{goal})$$

Druhá podmienka súvisí s priestorom v blízkosti cieľového bodu q_{goal}. Kvôli nespojitostiam v Gradiente v okolí koncového bodu môže dôjsť ku kmitaniu pohybu robota. Aby sme sa tomuto stavu vyhli, zavedieme si takú potenciálovú funkciu aby hodnota Gradientu pri približovaní sa k cieľu výrazne klesala. Príkladom takejto funkcie je funkcia rastúca kvadraticky so vzdialenosťou od cieľa. Po dosadení takejto funkcie si môžeme vzťah funkcie upraviť na:

$$U(q) = \frac{1}{2}\zeta d^2(q, q_{goal})$$

Príťažlivý Gradient tejto funkcie bude potom následovný:

$$\begin{aligned} \nabla U_{att}(q) &= \nabla(\frac{1}{2}\zeta d^2(q, q_{goal})), \\ &= \frac{1}{2}\zeta \nabla d^2(q, q_{goal}) \\ &= \zeta(q - q_{goal}) \end{aligned} \tag{2.1}$$

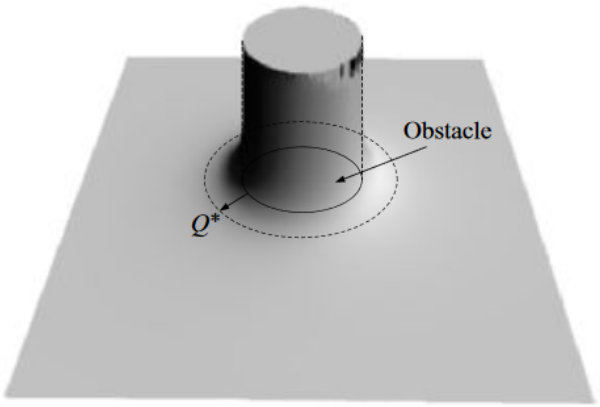
Takto zväčšujúca sa magnitúda vektoru vzhľadom ku vzdialenosti znamená, že ak je robot ďaleko od cieľa približuje sa rýchlejšie ako keď je k nemu blízko. V simulácii nám táto vlastnosť pomôže zaistiť vyššiu presnosť a zabráni kmitaniu, či nejasnému pohybu robota blízko cieľa pri prudkej zmene smeru. A pri reálnej aplikácii by táto vlastnosť pomáhala zabrániť roboti zabrzdiť až potom ako minie cieľ.

Odpudivý potenciál

Odpudivý potenciál budeme využívať, ak sa robot potrebuje vyhnúť nejakej prekážke. Ako som už v predošlej podkapitole načrtol, tento potenciál ho bude odpudzovať od danej prekážky. Sila potenciálu bude opäť závisieť na blízkosti robota k prekážke. Čím bližšie pri hrane, tým väčšia by mala byt magnitúda vektoru. Odpudivý potenciál je teda spravidla definovaný na základe vzdialenosti k najbližšej prekážke:

$$U_{rep}(q) = \begin{cases} \frac{1}{2}\eta(\frac{1}{D(q)} - \frac{1}{Q^*})^2 & , D(q) \leq Q^* \\ 0 & , D(q) > Q^* \end{cases}$$

Pre jednoduchšie pochopenie zobrazuje nasledujúci obrázok ten istý pomer:



Obr. 2.14: Okolie prekážky so vzdialenosťou Q^* ovplyvňuje robota [3]

Gradient tohto potenciálu potom bude daný vzorcom:

$$\nabla U_{rep}(q) = \begin{cases} \eta(\frac{1}{Q*} - \frac{1}{D(q)})\frac{1}{D^2(q)}\nabla D(q) & , D(q) \leq Q^* \\ 0 & , D(q) > Q^* \end{cases}$$

Kde $Q^* \in \mathbb{R}$ je faktor ktorý dovoľuje roboty ignorovať prekážky v dostatočnej vzdialenosti. η môžeme vnímať ako prírastok odpudivého Gradientu. Tieto skalárne veličiny sú učené ručne, bez výpočtu a spravidla sa určujú metódou pokus-omyl.

Gradientný zostup

Gradientný zostup sme si už nepriamo opísali v predchádzajúcej kapitole. Idea je veľmi jednoduchá, robot sa v prostredí pohybuje vždy v negovanom smere Gradientu. Pri každom pohybe sa robot ocitne v novej konfigurácii a tento proces sa opakuje až kým nedosiahneme bod, v ktorom je Gradient rovný 0. Pseudokód algoritmus by vyzeral následovne:

Algorithm 11 Gradientný zostup

```
1: q(0) = q_start
2: i = 0
3: while ∇ U(q(i)) ≠ 0 do
4:     q(i + 1) = q(i) + α(i) + α(i)∇U(q(i))
5:     i = i + 1
```

V tomto pseudo algoritme premenná $q(i)$ znázorňuje hodnotu q v $i-$tej iterácii. Výsledok toho algoritmu potom je sekvencie $q(0), q(1), ..., q(i)$. Hodnota $\alpha(i)$ predstavuje veľkosť kroku v $i-$tej iterácii. Túto veľkosť kroku je potrebné vhodne nastaviť pred začiatkom algoritmu, ale efektívne ju je potrebné meniť aj počas behu. Musí byť dostatočne malá aby náš robot "neskočil"do prekážky no zároveň pri pohybe voľným priestorom musí byť dostatočne veľká aby sa znížila výpočetná zložitosť. Hodnota α sa dá nastaviť napríklad na základe vzdialenosti k najbližšej prekážke. V prípadoch pohybu robota v priestore je tiež nepravdepodobné, že dosiahneme presný bod vyhovujúci podmienke ∇ U(q(i)) $\neq 0$. Túto podmienku teda v reálnom algoritme môžeme upraviť na tvar $||\nabla$ U(q(i))$|| < \epsilon$, kde ϵ znázorňuje vhodne malú odchýlku.

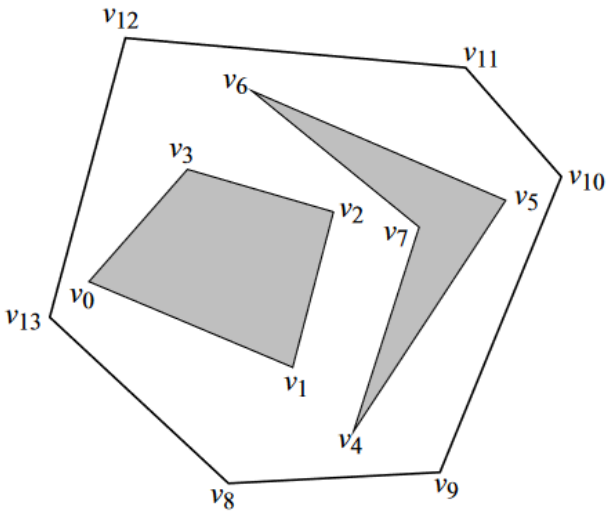
Bunečná dekompozícia

V tejto kapitole sa budem zaoberať iným druhom rozdeľovania voľného priestoru, a tým je *exaktná bunková dekompozícia*. Rozdelením priestoru takouto metódou získame konečný počet regiónov, takzvaných buniek. V rámci týchto buniek nás bude zaujímať prevažne ich stred (jadro) a steny bunky (hrany). Susedné bunky sú také bunky v priestore ktoré zdieľajú aspoň jednu hranu. Tieto hrany medzi bunkami môžu predstavovať fyzickú zmenu pre nášho robota v priestore, keďže v rámci jednej bunky prevažne hovoríme o homogénnom prostredí a nastavené globálne premenné budú platiť pre celú bunku. Ak však prejdeme z jednej bunky do druhej, môže sa zmeniť napríklad ukazovateľ na najbližšiu prekážku, ukazovateľ priamej cesty k iným prekážkam, či cieľu.

Graf priľahlosti je, ako je zrejmé už z názvu, graf spájajúci priľahlé bunky. Uzly v tomto grafe sú predstavené bunkami a hrany stenami jednotlivých buniek. Ak zdieľajú dve bunky rovnakú stenu, bude uzly reprezentujúce tieto bunky v grafe spájať aj hrana. Vytváraním takéhoto grafu priľahlosti dokážeme eventuálne nájsť cestu z počiatočného bodu do bodu koncového. Do grafu sa najprv pridajú bunky obsahujúce štart a cieľ a potom sa pridávaním uzlov hľadá cesta spájajúca tieto dva uzly. Metóda vytvárania takéhoto grafu môže taktiež poslúžiť na mapovanie celého priestoru. Poznáme viac typov bunečnej dekompozície, v tejto práci sa budem zaoberať prevažne **lichobežníkovou** dekompozíciou.

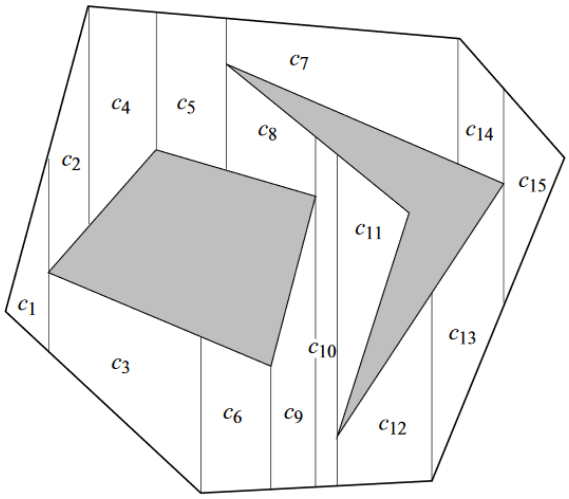
Lichobežníková dekompozícia

Lichobežníková bunečná dekompozícia je pomerne jednoduchá. Predpokladá však mnohouholníkový tvar skúmaného priestoru a nie je preto všeobecná. Pri tejto metóda vytvárame bunky lichobežníkového prípadne trojuholníkového tvaru. Trojuholník môžeme vnímať ako degradovaný lichobežník s jednou stranou nulovej dĺžky a teda spadá do tejto metódy.
Predpokladajme mnohouholníkový voľný priestor s prekážkami taktiež mnohouholníkového tvaru. Pre zjednodušenie demonštračných účelov predpokladajme, že každý vrchol týchto polygónov má unikátnu $X-$ovú súradnicu. Každý z týchto vrcholov V označíme unikátnym indexom i Vizualizácia tohto priestoru môže vyzerať následovne:



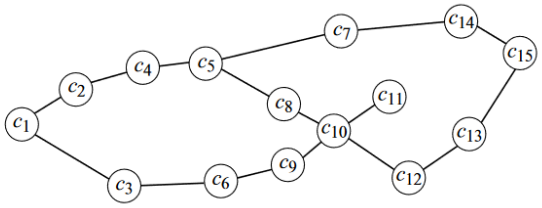
Obr. 2.15: Označenie vrcholov unikátnym identifikátorom [3]

Aby sme nad takýmto priestorom previedli lichobežníkovú dekompozíciu, z každého vrcholu V_i pre $i \in 0 - N$ kde N značí počet vrcholov, vyvedieme dve úsečky. Úsečka smerujúca smerom hore, nazývaná *vrchné vertikálne rozšírenie* smeruje po ose Y s inkrementujúcou hodnotou a *spodné vertikálne rozšírenie* smeruje po ose Y s dekrementujúcou hodnotou. Úsečka s počiatočným bodom vo vrchole bude na druhej strane ukončená prvou hranou na ktorú narazí, a teda buď hranou inej prekážky alebo ohraničením priestoru. Tieto úsečky vytvoria uzatvorené oblasti ktoré nazveme *bunky*. Po vytvorení buniek a ich očíslovaní bude náš predošlí priestor vyzerať následovne:



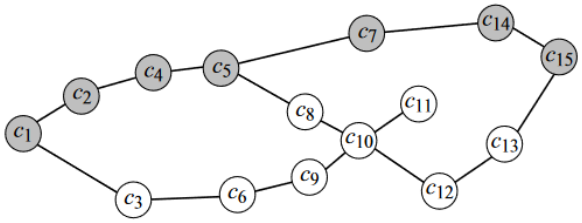
Obr. 2.16: Vytvorenie vertikálnych rozšírení a očíslovanie vzniknutých buniek [3]

Ako je opísané v predošlej podkapitole, nad týmto priestorom vieme vytvoriť graf priľahlosti. Vizualizovaný graf potom bude vyzerať následovne:



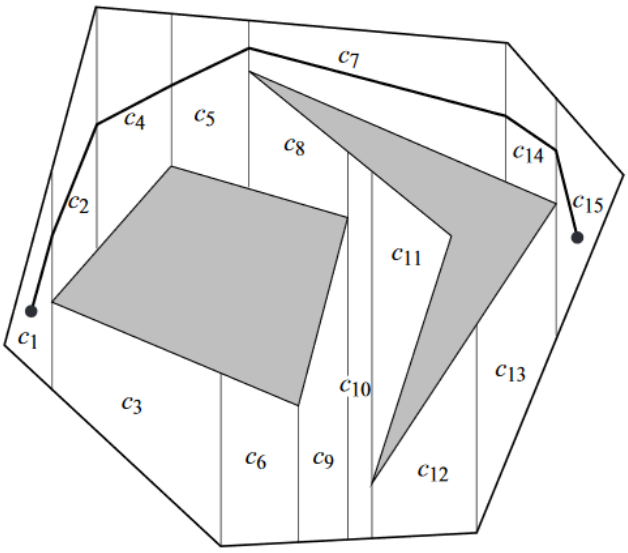
Obr. 2.17: Vytvorenie grafu priľahlosti z predošlého priestoru [3]

Teraz môžeme pomocou algoritmu skúsiť nájsť cestu týmto grafom priľahlosti. Najprv musíme teda určiť v ktorých bunkách sa nachádza štartovný a cieľový bod. Pre tento príklad si môžeme určiť, že počiatočný bod sa nachádza v bunke C_1 a koncový bod v bunke C_{15}. Najkratšia cesta grafom priľahlosti potom bude:



Obr. 2.18: Nájdenie cesty v grafe priľahlosti z bunky C_1 do bunky C_{15} [3]

Nájdenie takejto trasy ešte stále nezaručuje súvislú prechodnú cestu pre nášho robota. Keďže spájame len bunky ako celky, priame cesty z jadra do jadra by mohli viesť cez hrany prekážok. Musíme preto do plánovanie cesty zapojiť aj samotné hrany buniek, a to následovne: keďže trapezoid je konvexný útvar, každé dva body v jeho vnútri môžu byť spojené bez obavy, že budú prechádzať nejakou prekážkou. Túto vlastnosť môžeme využiť tak, že namiesto spájania jadier buniek budeme spájať *stredy* susedných stien jednotlivých buniek. Takto nám vznikne neprerušovaná cesta grafom bez stretu s prekážkou. Po prevedení tohto kroku môžeme cestu priestorom vizualizovať následovne:



Obr. 2.19: Znázornenie cesty priestorom pri využití spájanie stredov susedných hrán buniek [3]

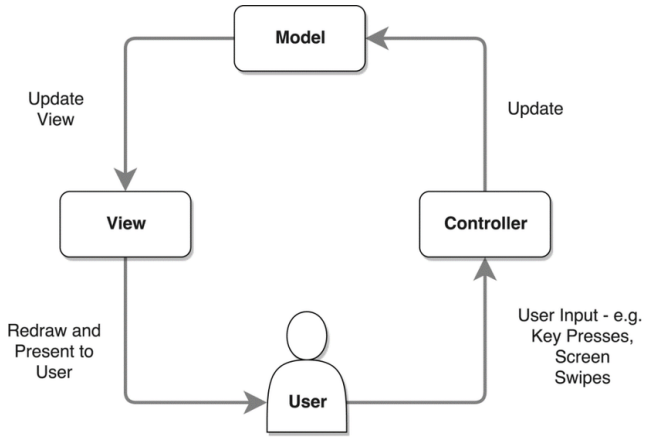
Kapitola 3

Demonštračná aplikácia

3.1 Architektúra

Všetky vizualizácie sa budú vykonávať v Java aplikácii vytvorenej za použitia knižnice vizlib Jakuba Rusnáka vytvorenej pre jeho diplomovú prácu. Aplikácia implementuje vzor Model-View-Controller a teda efektívne oddeľuje aplikačnú logiku od zobrazenia pre užívateľa.

Pri prenesení vzoru Model-View-Controller do demonštračnej aplikácie model predstavuje algoritmus s dátovými štruktúrami a logikou a ten sa reprezentuje pre užívateľa pomocou užívateľského rozhrania.



Obr. 3.1: Zobrazenie modelu model-view-controller [6]

V pôvodnom návrhu Jakuba Rusnáka sa pre každý algoritmus vytvárala samostatná aplikácia. K tejto práci je priložená len jedna aplikácia obsahujúca všetky algoritmy vytvorené Jakubom Rusnákom takisto ako aj nové algoritmy. Takáto aplikácia nepotrebuje žiadne ďalšie prostredie na navigáciu medzi algoritmami (pôvodne bolo využívané prostredie webovej stránky PHP).

Základovým kameňom aplikácie je objekt typu JFrame ako nosič panelu jednotlivého algoritmu. Pre tento JFrame je vytvorených n panelov typu JPanel, kde n je počet algoritmov implementovaných v aplikácii. Pre každý algoritmus teda existuje práve jeden panel ktorý sa momentálne zobrazuje pre užívateľa. Konkrétny panel potom spĺňa úlohu ovládacieho panelu algoritmu a reaguje na zmeny vytvorené užívateľom alebo samotným algoritmom. Pri každom prepnutí algoritmu na iný sa odstráni spojenie sledovania akcií starého pa-

nelu a pridá nové. Efektívne teda po prepínaní algoritmov dochádza ku komunikácií len príslušného algoritmu, jeho ovládacieho panelu a modelov.

Každý ovládací panel JPanel spúšťa jeden algoritmus. Táto konštrukcia je ponechaná hlavne z toho dôvodu, že každý algoritmus môže využívať rozdielnu logiku a dáta a preto pri špecifických stavoch alebo obnove stavu algoritmu je potrebné vykonať rôzne akcie. Jeden ovládací panel pre všetky algoritmy by bol teda zbytočne zložitý a neprehľadný, či dokonca nefunkčný.

Pre každý ovládací panel a teda aj pre každý algoritmus sú v zdrojových súboroch špecifikované pohľady. Každý algoritmus môže využívať kombináciu nasledovných :

- **CodeView** - pseudokód zobrazený pre lepšie pochopenie funkčnosti algoritmu
- **ConsoleView** - miesto pre výpisy algoritmu - nájdenie cesty, oznámenie nedostupnosti cesty alebo zaujímavé či potrebné informácie o premenných v algoritme
- **DrawView** - zobrazovacie prvky simulácie
- **MapView** - konfiguračný priestor algoritmu. Ráta s pridaním práve jedného štartu, cieľa a ľubovoľného počtu prekážok. Umožňuje pridávanie, mazanie alebo menenie tvaru prekážky.
- **ParameterView** - umožňuje užívateľovi zadať vstupné hodnoty algoritmu ako napríklad maximálny počet uzlov pri expanzívnych stromoch alebo dohľad senzora tangent bug algoritmu. Takisto ponúka sadu preddefinovaných konfigurácií pre rýchlejšie spustenie vizualizácie algoritmu.
- **StatusBarView** - umožňuje výpis stavu algoritmu a ponúka aj progress-bar pre ľahšie predstavenie pokročenia algoritmu
- **TextView** - jednoduché textové pole pre zobrazenie informácií pre užívateľa. Dá sa využiť pri zložitejších algoritmoch alebo pri špecifických situáciach na objasnenie správania algoritmu či poukázanie na určitý jav.
- **ToolbarView** - ovládací panel simulácie, ponúka radu tlačítok na krokovanie, spúšťanie či zastavenie alebo resetovanie simulácie algoritmu.
- **GridMapView** - konfiguračný priestor algoritmu rozdelený do štvorcových buniek v tvare mriežok, umožňuje meniť veľkosti mriežok.
- **GridMapSetView** - pri algoritmoch využívajúcich množiny, ako napríklad množina open a closed pri algoritme BFS, umožňuje ich zobrazenie užívateľovi.

Výhodou vytvárania a pridávania algoritmov pri použití tejto knižnice na tvorbu aplikácie je zredukovanie potreby programátora prispôsobovať rozhranie užívateľovi. Pohľady opísané vyššie môžeme vnímať podobne ako šablóny alebo formy, do ktorých programátor napasuje vlastný algoritmus a ten sa bude ovládaním a vzhľadom podobať na všetky ostatne algoritmy v danej aplikácii. Užívateľovi táto vlastnosť umožní rýchlejšie pochopenie algoritmu, keďže nebude musieť pri novom algoritme spoznávať aj nové prostredie a ovládanie.

3.2 Vstupné parametre

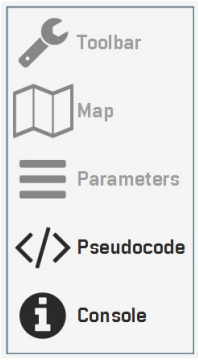
Všetky vstupné parametre zadané užívateľom v pohľade **parameterView** sú automaticky po zadaní uložené do globálnych vlastností aplikácie *Properties*. Všetky hodnoty sú potom dostupné z celej aplikácie ako napríklad v kóde 3.1, kde sa v algoritme využíva hodnota veľkosti tela robota.

```
robotSize = Properties.INSTANCE.getPropertyInteger(TangentBugParameters.
    RobotSize);
```

Výpis 3.1: Získanie hodnoty veľkosti robota pre algoritmus tangent bug z globálnych vlasností *Properties*

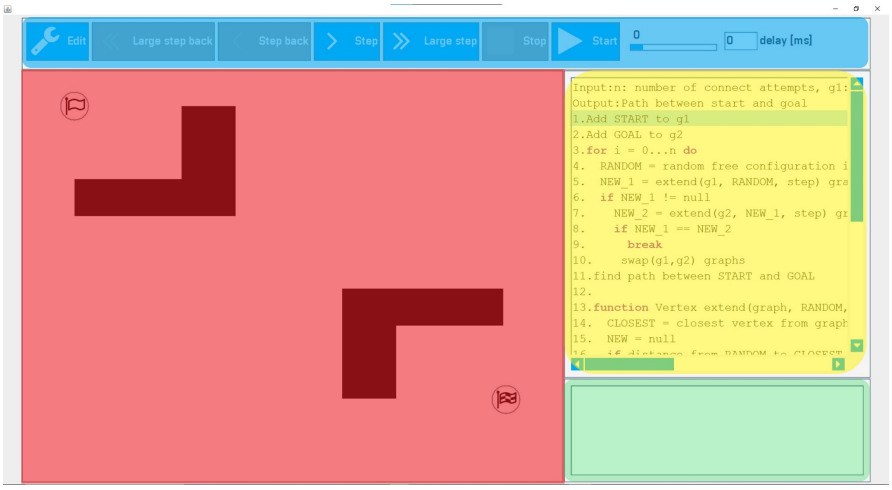
3.3 Užívateľské rozhranie

V tejto sekcii je znázornená vizualizácia algoritmu, aby čítajúci mohol vizualizovať algoritmy bez rozsiahleho študovania funkcionality aplikácie. Aplikácia sa skladá z kombinácie pohľadov popísaných v predošlej kapitole a pre príklad užitia aplikácie bol vybraný algoritmus RRT, ktorý využíva 4 základné z nich 3.4. Veľkosť týchto pohľadov nie je pevná a užívateľ si ich môže prispôsobovať podľa svojej potreby a preferencie 3.3. Pri kliknutí pravým tlačítkom myši na ľubovoľný pohľad sa užívateľovi zobrazí menu s dostupnými pohľadmi ktoré si môže na tomto mieste zobraziť 3.2.



Obr. 3.2: Manipulácia zobrazení. Prevzaté z [3]

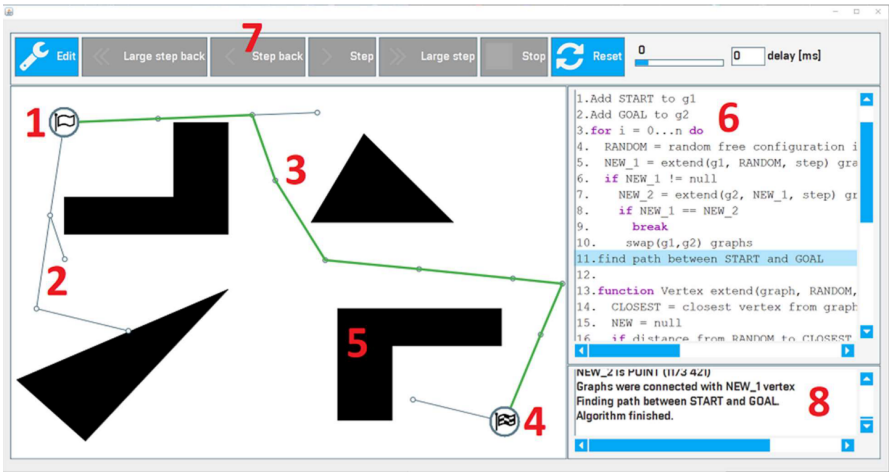
Obr. 3.3: Manipulácia zobrazení. Prevzaté z [3]



Obr. 3.4: Rozdelenie zobrazenia aplikácie do zón - jednotlivých pohľadov [5]

- červená zóna - konfiguračný priestor algoritmu
- žltá zóna - pseudokód algoritmu pre pochopenie princípov
- zelená zóna - konzola pre vypisovanie potrebných či zaujímavých hodnôt z výpočtu algoritmu
- modrá zóna - ovládanie simulácie, podrobný popis v 3.3.1

V jednotlivých pohľadoch sa potom nachádzajú konkrétne informácie potrebné či zaujímavé pre užívateľa. Na obrázku 3.5 je použitý algoritmus RRT vytvorený Jakubom Rusnákom.



Obr. 3.5: Očíslované sekcie vizualizačného nástroja Jakuba Rusnáka [5]

- 1. Počiatočná pozícia robota
- 2. Cesta vygenerovaná algoritmom

- 3. Cesta vybraná algoritmom pre pohyb robota
- 4. Cieľová pozícia
- 5. Prekážka
- 6. Zobrazený pseudokód použitého algoritmu
- 7. Prepínanie medzi editačným a simulačným módom a ovládanie
- 8. Konzola s výpisom

3.3.1 Ovládanie simulácie

Simulácia sa ovláda pomocou tlačidiel v toolbare, spravidla v hornej časti okna. Tento toolbar sa môže nachádzať v dvoch módoch, simulačnom 3.6 a editovacom 3.7. Pri simulačnom móde ovládame simuláciu a zobrazujeme jednotlivé kroky ktoré algoritmus podstúpil a pri editačnom nastavujeme konfiguračný priestor pridávaním, presúvaním či mazaním prekážok a presúvaním počiatočnej a koncovej konfigurácie.

Obr. 3.6: Tlačítka ovládania simulácie [5]

- Edit - prepínanie medzi editačným a simulačným módom
- Large step back - vráti simuláciu o jeden veľký krok dozadu, spravidla jedna podmienka alebo celý cyklus while
- Step back - vráti simuláciu o jeho naposledy vykonanú činnosť dozadu
- Step - posunie simuláciu o najbližšiu vykonanú činnosť
- Large step - preskočí cyklus alebo podmienku v simulácii
- Stop - zastaví výpočet algoritmu a ponúkne tlačítko reset na znovuobnovenie simulácie
- Start - spustí simuláciu v zadanej rýchlosti, pri opätovnom stlačení simuláciu pozastaví
- Delay - nastaví čakaciu dobu medzi každým krokom simuláciu

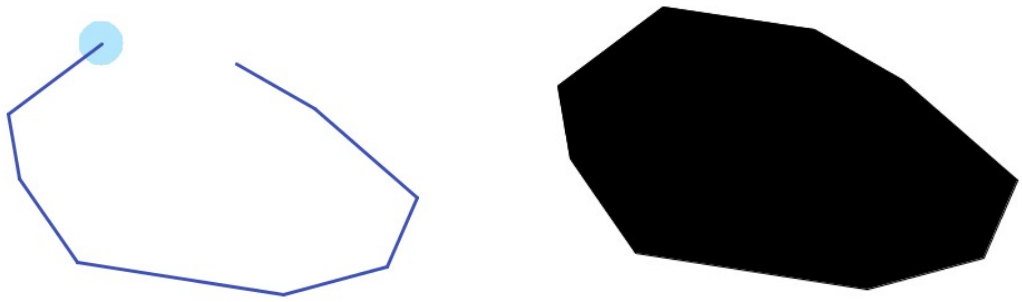
Obr. 3.7: Tlačítka nastavovania konfiguračného priestoru [5]

- Simulation - prepínanie medzi editačným a simulačným módom
- Start - pridá do konfiguračného priestor počiatočnú konfiguráciu robota

- Goal - pridá do konfiguračného priestor koncovú konfiguráciu robota
- Obstacle - pridá novú prekážku do konfiguračného priestoru, podrobnejší opis pridania 3.3.1
- Remove - po kliknutí na prekážku ju odstráni z konfiguračného priestoru

Pridanie prekážky

Po zakliknutí tlačidla pridania prekážky môže užívateľ pridať mnohouholník klikaním do konfiguračného priestoru a tým pridávaním nových vrcholov tohto mnohouholníku. Po znovu kliknutí na počiatočný vrchol sa prekážka uzatvorí a pridá 3.8.



(a) Pridávanie nových vrcholov prekážky

(b) Po zakliknutí na pôvodný vrchol sa prekážka vytvorí a pridá do konfiguračného priestoru

Obr. 3.8: Pridávanie novej prekážky v tvare mnohouholníku do konfiguračného priestoru

3.4 Simulácia

Počas výpočtu algoritmu sa v podstatných bodoch vizualizácie volá funkcia **addCommand** ktorá nastaví stav vizualizácie v danom bode. Každé zavolanie tejto funkcie potom predstavuje jeden okamih simulácie a na konci výpočtu algoritmu sa započne simulačná fáza, kde si užívateľ zobrazuje funkčnosť´t algoritmu, zavolaním **simulate(getCommands());**.

3.5 Vytváranie nových algoritmov

Vloženie nového algoritmu do existujúcej šablóny je pomerne jednoduché. Ovládacie prvky a prvky GUI sa podľa základného návrhu nachádzajú v súboroch **Frame.java** a **MainFrame.java**. Samotný program algoritmu je potom pripojený v samostatnom .java súbore a je tak efektívne oddelený od GUI. Po vytvorení funkčného algoritmu v jazyku java je potrebné pridať vizualizačné prvky do zložky 'views'. Táto zložka bude obsahovať pseudokód, ovládanie konzolového výstupu, predom nahraté mapy prostredia pre daný algoritmus, zdrojový kód pre vyznačovanie nájdenej cesty a toolbar s ovládacími prvkami aplikácie.

Podrobnejší návod na pridanie nového algoritmu je popísaný v prílohe C.

Po pridaní zdrojových súborov nového algoritmu stačí aplikáciu jednoducho preložiť a spustiť, a to buď debugovacom móde, alebo vytvoriť samostatne stojacu java aplikáciu.

Kapitola 4

Implementácia bug algoritmov

Všetky tri bug algoritmy sa skladajú z piatich pohľadov vytvorených za pomoci knižnice vizlib Jakuba Rusnáka. Konkrétne sú to pohľady:

- Code view
- Console view
- Map view
- Parameter view
- Toolbar view

Opis jednotlivých pohľadov je dostupný v kapitole 3.1. Ďalej je každý algoritmus uložený v súbore nazovAlg.java a každý algoritmus je ovládaný vlastným hlavným panelom, tak ako je opísané v kapitole 3.1.

Po zavolaní funkcie **runnable** v zdrojovom kóde každého bug algoritmu sa ako prvé kontroluje výskyt počiatočnej a koncovej konfigurácie v priestore. Na toto si využívajú funkcie vstavané do knižnice vizlib **map.getStart()** a **map.getGoal()**. Obe tieto konfigurácie musia byť nastavené a zároveň nesmú ležať v prekážke, inak algoritmus zahlási chybu a výpočet nezapočne. Následne sa pre každú prekážku zavolá funkcia na rozšírenie prekážky. Princíp a dôvod rozširovania je popísaný nižšie, v kapitole 4.2.

Pomocou funkcie **addCommand** sa potom vytvorí prvý krok simulácie pohybu robota. Princíp a použitie tejto funkcie je presnejšie popísané v architektúre aplikácie 3.4.

```
addCommand(new UndoableCommand(){
  @Override
  public void redo(){
    graph.addVertex(robot);
    changeUI(1, "Adding a robot." + map.getStart().getCoordinate().x + "y:"
        + map.getStart().getCoordinate().y, null, "RANDOM");
  }

  @Override
  public void undo(){
    map.removeObject(robot);
    changeUI(1, "Removing robot. " + null, null, "RANDOM");
  }
```

```
});
```

Výpis 4.1: Pridanie robota do grafu uloženého v globálnych parametroch pre zobrazenie užívateľovi.

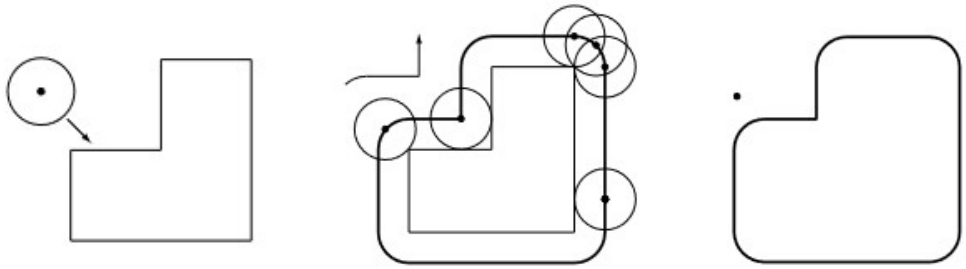
Na konci funkcie **runnable** každý z algoritmov zavolá funkciu **move**, ktorá začne hýbať robotom smerom k cieľu. Aby sa zlepšil výkon aplikácie, pozícia robota sa pre užívateľa obnovuje len každú dvadsiatu iteráciu pohybu, s krokom o veľkosti 0.5 koordinátu. Na obnovenie zobrazenia robota sa využíva opäť funkcia **addCommand**. Každé zavolanie je teda vo výsledku jeden krok simulácie. Algoritmus sa cyklí vo funkcii **move** až kým robot nenarazí na cieľ alebo kým sa neodhalí prekážka. Každý algoritmus odhlanie prekážky implementuje inak a konkrétne funkcie sú popísané v v sekciách 4.4, 4.5 a 4.6

4.1 Zobrazenie robota

Na vytvorenie objektu robota bol vytvorený objekt **robotBody** ktorý rozširuje objekt **SimpleVertex** nachádzajúci sa v knižnici vizlib vytvorenej Jakubom Rusnákom. Tento nový objekt spája jednoduchosť určenia pozície jednorozmerného bodu SimpleVertex a pripája k nemu nastaviteľnú ikonu robota pre vizualizáciu jeho veľkosti v priestore.

4.2 Určenie konfiguračného priestoru robota

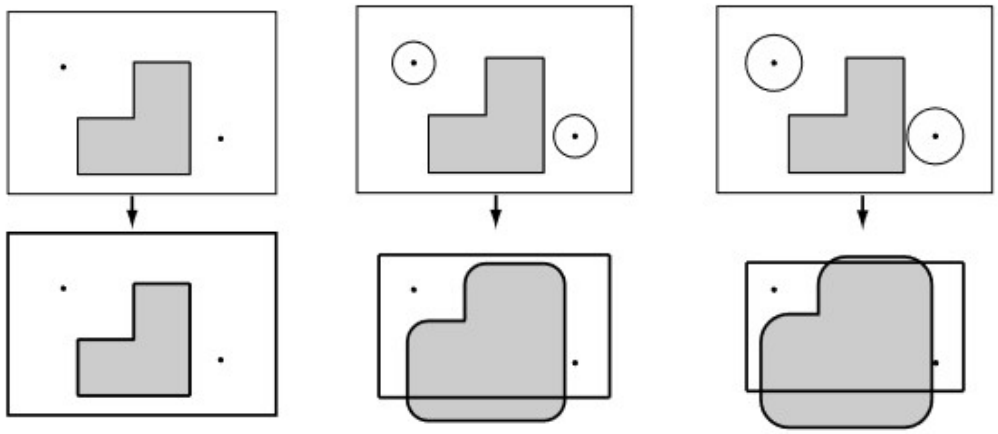
Ďalším problémom je simulácia tela robota. V implementácii aplikácie opisovanej touto prácou je náš robot jednorozmerný bod, a preto je potrebné nie len vizualizovať, ale aj simulovať jeho šírku a dĺžku, aby simulácia mohla odpovedať pohybu skutočného robota (ten nebude nikdy jednorozmerný). Jednoduchšie ako zväčšovať telo robota a komplikovane hľadať priesečníky s prekážkami, v simulácii môžeme zvoliť opačný prístup a namiesto robota zväčšiť prekážky. Pre každú prekážku 4.4a vytvoríme prekážku rozšírenú o buffer (bez slovenského prekladu - zóna fixnej vzdialenosti od objektu) a vložíme novú prekážku do konfiguračného priestoru - obrázok 4.4b. Takto efektívne ovplyvníme konfiguračný priestor robota v závislosti na jeho veľkosti. Rozšírenie jednej prekážky v tvare mnohouholníku si môžeme predstaviť nasledujúcim spôsobom 4.1: presunieme pomyselného robota na hranu prekážky tak, že sa prekážky vždy dotýka no nikdy sa nenachádza v jej vnútri 4.1a a robota presúvame postupne okolo celého obvodu 4.1b. Počas tohto pohybu mapujeme pozíciu stredu robota a spájaním všetkých pozícií stredu vytvoríme novú prekážku efektívne zväčšenú o polovicu veľkosti tela robota 4.1c.



(a) (b) (c)

Obr. 4.1: Opis zväčšenia mnohouholníkovej prekážky pre aktualizáciu konfiguračného priestoru robota v troch krokoch

Čo to znamená pre nášho robota a jeho konfiguračný priestor? V nasledujúcom obrázku sú zobrazené 3 roboty, jeden s nulovou veľkosťou (bodový robot) 4.2a, jeden robot s kruhovým telom 4.2b a väčší robot s kruhovým telom 4.2c. Pre každú situáciu bol upravený konfiguračný priestor (vonkajší obdĺžnik) zväčšením prekážky metódou popísanou vyššie.



(a) (b) (c)

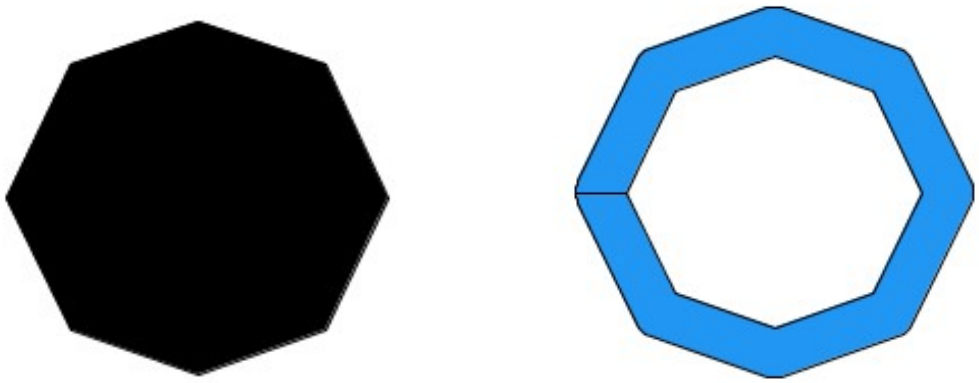
Obr. 4.2: Upravenie pôvodného (hore) priestoru na konfiguračný priestor (dole) v závislosti na veľkosti robota. Robot s nulovou veľkosťou (bodový robot) 4.2a, robot s kruhovým telom 4.2b a väčší robot s kruhovým telom 4.2c

Pri implementácií je táto metóda pomerne náročný proces a v praktickej časti tejto práce ju nahradila funkcia *buffer* z Java knižnice *com.vividsolutions.jts.geom.Geometry* ktorá vo výsledku vytvára rovnakú prekážku a teda aj rovnaký konfiguračný priestor. Funkciu použijeme na vytvorenie prekážky objektu **obstacleBuffer**, ktorú som vytvoril pre potreby nastavenie farby výplne prekážky. Tento objekt rozširuje objekt **Obstacle** z knižnice vizlib. Po zavolaní funkcie na prekážku 4.3a dostaneme prekážku 4.3b. Aby sme dostali plnú prekážku, musíme odstrániť vnútorné vrcholy rozšírenej prekážky. Toto môžeme urobiť kontrolovaním, či sa niektorý z vrcholov nachádza v súradniciach pôvodnej nerozšírenej prekážky a ak áno, odstránime ho z novej prekážky. Ak s novým súborom vrcholov vytvoríme novú prekážku objektu **obstacleBuffer**, dostávame vyplnenú, zväčšenú modrú prekážku. Aby

sme ale užívateľovi zobrazil rozdiel medzi starou a novou prekážkou a aby simulácia pôsobila krajšie, pridáme na túto novú prekážku aj prekážku starú v čiernej farbe, vznikne nám tak modrý okraj naznačujúci o koľko sa prekážka rozšírila a kade sa bude náš robot pohybovať 4.4.

```
for(PaintableShape obstacle : map.getObstacles()){
    obstacleBuffer buffer = (new obstacleBuffer(Arrays.asList(obstacle.
        getAsGeometry().getBoundary().buffer(robotSize/2).getCoordinates())
        ,1));
```

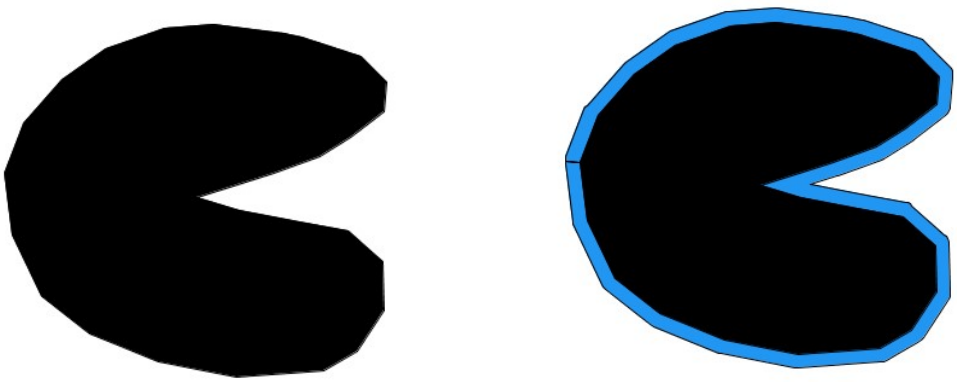
Výpis 4.2: zošírenie prekážky - buffering



(a) Prekážka v konfiguračnom priestore

(b) Vytvorenie bufferu prekážky

Obr. 4.3: Vytvorenie prekážky objektu **obstacleBuffer**.



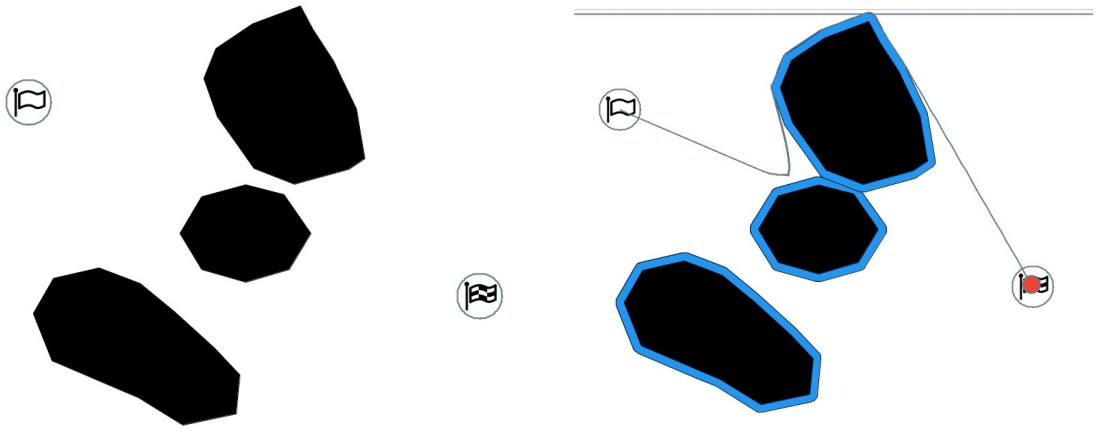
(a) Prekážka v konfiguračnom priestore

(b) Rovnaká prekážka rozšírená o buffer

Obr. 4.4: Skombinovanie pôvodnej prekážky v priestore s jej bufferom a vytvorenie rozšírenej prekážky s okrajom inej farby pre lepšiu vizualizáciu

Samotnému robotovi potom pridáme vzhľad a veľkosť, no s týmto objektom prakticky nerobíme žiadne operácie okrem jeho posunu na obrazovke pre zobrazenie užívateľovi.

na obrázku 4.5 je znázornená situácia kedy nám rozšírenie prekážok pomohlo odhaliť príliš úzku medzeru medzi dvoma prekážkami.



(a) Prekážky v konfiguračnom priestore

(b) Rovnaké prekážky rozšírené o buffer

Obr. 4.5: Princíp simulácie veľkosti robota - rozšírenie prekážok v konfiguračnom priestore a vplyv rozšírenia na robota využívajúceho algoritmus Tangent Bug

4.3 Určenie smeru obchádzania prekážky

Bug a Bug II algoritmy

V definícii Bug a Bug II algoritmov sa uvádza, že smer obchádzania prekážok je predom daný. Keďže sú takéto roboty vybavené len dotykovým senzorom, určovanie uhlu narazenia do prekážky by bolo buď komplikované a neisté, alebo nemožné. V mojej implementácii si je užívateľ schopný nastaviť smer obchádzania prekážok pomocou prepínača v parametroch 4.6.

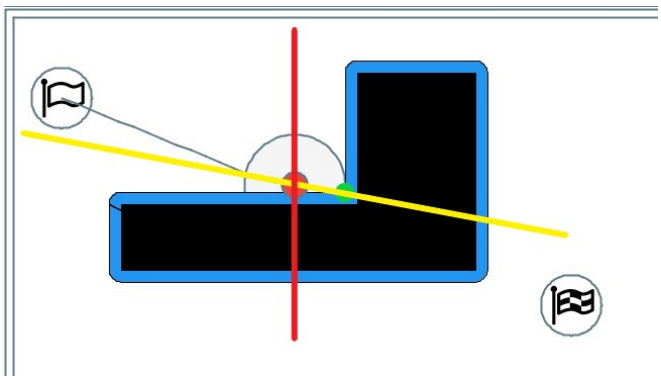
Obr. 4.6: Prepnutie smeru obchádzania prekážky v algoritmoch Bug a Bug II

Tangent Bug

Pre potreby Bug algoritmov je však potrebná aj simulácia stretu s prekážkami a teda pre výpočet trasy robota je potrebné nájsť priesečníky s týmito prekážkami. Pri Bug I a Bug II algoritmoch je aj táto implementácia priamočiara. Pri využití funkcie na porovnanie jednoduchých 2D súradníc vieme určiť, či sa robot pretína s nejakou prekážkou a potom ďalej reagovať na túto skutočnosť.

Pri algoritme Tangent Bug je však potrebné simulovať senzory s určitou dĺžkou. Tieto

senzory sú v tejto práci simulované pomocou úsečiek. Ich fixnú dĺžku nastavuje užívateľ pred spustením algoritmu.
Pri algoritme tangent bug je potrebné zistiť uhol pod ktorým narazil robot na prekážku, respektíve uhol senzoru ktorý prekážku objavil. Kým robot obchádza prekážku prirodzene (posúva sa k bodu najbližšiemu k cieľu) netreba uhol rátať, no keď robot nájde lokálne minimum, je potrebné pokračovať v rovnakom smere obchádzania ako sa do toho momentu robot pohyboval. V implementácii som si preto vytvoril úsečku spájajúcu robota a lokálne minimum prekážky (predĺžená žltá čiara) 4.7. Podľa tejto čiary a uhlu nájdenia prekážky jednoducho určíme ku ktorým stretom simulovaných senzorov s prekážkou sa bude robot pohybovať.



Obr. 4.7: Zobrazenie určenia smeru obchádzania prekážky pri nájdení lokálneho minima algoritmom Tangent Bug

4.4 Implementácia Bug

Bug algoritmus simuluje robota s taktilným senzorom, teda prekážku odhalí až keď na ňu narazí simulovaný objekt robota. Na toto odhalenie sa používa pri každej iterácii pohybu funkcia **isInObstacle** z knižnice vizlib. Ak robot narazí na prekážku, zavolá funkciu **circleObstacle**, ktorá bude simulovať pohyb robota okolo prekážky. V každom podstatnom bode algoritmu je zavolaná funkcia **addCommand** na pridanie simulačného kroku pre užívateľa.

Vo funkcii na obchádzanie prekážky sa využívajú koordináty vrcholov prekážky získané z objektu prekážky na ktorú robot narazil. Z týchto vrcholov sa následne vytvorí množina hrán prekážky, po ktorých sa náš robot bude pohybovať. Vďaka číslovaniu vrcholov sa vieme presúvať po jednotlivých hranách a uzatvorenie celého okruhu okolo prekážky môžeme teda kontrolovať číselne a nie porovnávaním súradníc, čím sa zbavíme prípadnej nepresnosti pri pohybe.

Pri každej iterácii avšak zaznamenávame polohu roboti voči cieľovej konfigurácii a pri celom okruhu nájdeme bod na hrane prekážky ktorý je k nej najbližšie, nazvime ho bod q. Počas krúženie taktiež zaznamenávame prejdenú vzdialenosť jednou stranou k bodu q a potom môžeme rozhodnúť ktorou stranou sa k tomuto bodu vrátiť (vyberieme kratšiu)

Po dosiahnutí bodu q sa opäť volá funkcia **Move**.

4.5 Implementácia Bug 2

Až do bodu krúženia okolo prekážky je implementácia algoritmu Bug 2 rovnaká ako Bug 4.4 až na rozdiel pridania úsečky m spájajúcej počiatočnú konfiguráciu s cieľovou. Pri pohybe okolo prekážky vo funkcii **circleObstacle** však nehľadáme bod najbližší cieľu, ale kontrolujeme, či ďalší pohyb robota nekoliduje s úsečkou m. Na toto používame pri každej iterácii pohybu metódu **intersects** objektu vytvoreného za pomoci knižnice vizlib. V prípade nájdeného priesečníku sa robot v ďalšej iterácii presunie na tento priesečník, a v prípade, že ešte tento bod nie je pridaný v množine **visited**, volá funkciu **move**. Ak je však tento bod v **visited**, algoritmus ho ignoruje a pokračuje v obchádzaní prekážky. Príklad nutnosti ignorovať bod opustenia prekážky je zobrazený v sekcii testovania 5.5.

4.6 Implementácia Tangent Bug

Tangent Bug algoritmus sa od Bug algoritmov líši hlavne tým, že nie je vybavený taktilným senzorom, ale senzorom s určitou vzdialenosťou. Pre pohybovaní robotom vo funkcii **move** bola preto pridaná kontrola odhalenia prekážky stojacej v ceste robota. Úsečka predstavujúca senzor S vždy smeruje od robota smerom k cieľovej konfigurácii (čo je smer pohybu robota vo voľnom priestore). Ak sa táto úsečka pretne s nejakou prekážkou, musí na ňu robot zareagovať, keďže do nej narazí. V tomto momente sa volá funkcia **circleObstacle**, ktorá riadi obchádzanie prekážky.

Vo funkcii **circleObstacle** sa využíva kruh *vicinity*, ktorého polomer je vždy rovný dĺžke senzora prebranej z parametrov algoritmu zadaných užívateľom. Body na hrane prekážky, kde sa *vicinity* pretína s obchádzanou prekážkou sú potom porovnávané a vyberá sa ten, ktorý je najbližšie cieľu. V momente, keď sa robot pohybuje k takémuto bodu a žiadne nové body neznižujú vzdialenosť k cieľu, algoritmus volá funkciu **followBoundary**.

Na začiatku funkcie **followBoundary** sa využijú hodnoty predané z funkcie **circleObstacle** a určí sa smer obchádzania prekážky. Vytvorí sa úsečka od robota smerom ku hrane prekážky na ktorú robot narazil a určí sa uhol tejto úsečky a hrany ako bolo znázornené na obrázku 4.7. Podľa toho sa určí index vrcholu prekážky ku ktorému by mal robot postupovať a teda aj smer obchádzania. (Ak je index ku ktorému má robot vrcholu väčší ako index na druhej strane tej istej hrany - cyklus vrcholov sa inkrementuje a naopak).

Kapitola 5

Testovanie implementácie

5.1 Prednastavené konfiguračné priestory - Preset

Pre každý algoritmus sa nachádza v pohľade parametrov množina predpripravených máp vytvorených programátorom pre rýchlejšie spustenie vizualizácie užívateľom. Pri náučnej aplikácií je veľmi podstatné aby algoritmy fungovali v týchto priestoroch tak ako majú. Pre každý algoritmus boli teda ručne testované všetky jeho prednastavené presety.

5.1.1 Pravdepodobnostné algoritmy

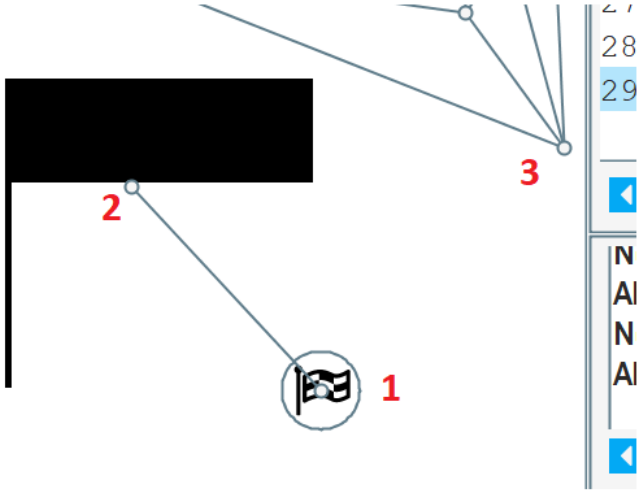
Pri pretváraní aplikácií Jakuba Rusnáka [3] do podoby odovzdávanej v tejto práci bol kladený vyšší dôraz na testovanie a prípadnú korekciu pravdepodobnostných algoritmov. Testovaniu predchádzalo štúdium funkcií a postupu týchto algoritmov a následný návrh „okrajových“ stavov pre tieto algoritmy (napríklad nedostupnosť cieľového bodu, príliš veľké či malé vstupné hodnoty pre algoritmus). Práve tieto okrajové stavy často rozlišujú jednotlivé algoritmy a správna simulácia ich funkčnosti a adekvátne zobrazenie ich správania je pri náučnej aplikácii dôležité. Zároveň je pomerne kontraproduktívne rozširovať aplikáciu o ďalšie algoritmy ak sa v nej nachádzajú už prípadne chybné algoritmy.

Nebudem v tejto práci vypisovať preklepy, gramatické chyby ani chyby ktoré budem považovať za kozmetické. Algoritmy a aplikáciu som analyzoval nasledujúcim spôsobom: najprv v roli študenta snažiaceho sa pochopiť algoritmus z vizualizačnej časti pomocou zobrazovacieho panelu a pseudokódu, neskôr podrobnejšie preskúmanie kódu v jazyku java. V následných odstavcoch sú zaznamenané moje poznatky.

PRM

Pripájanie cieľa a štartu do grafu
Hraničný stav: pomerne nelogická a nepredpokladaná situácia. Pre všetky vrcholy sa v algoritme PRM vytvára presne n počet hrán ktoré sa prípadne pridávajú do grafu konfiguračného priestoru. Pri pripájaní cieľovej a počiatočnej konfigurácie sa však pridáva 0 až n hrán do grafu spôsobom, že sa pridá vždy najkratšia možná hrana. Bod Q_{start} označujúci počiatočný bod a bod Q_{goal} označujúci koncový bod sa nespájajú s k najbližšími bodmi v grafe. Spájajú sa konkrétne s najbližším a toto môže viesť ku zlyhaniu algoritmu v momente, kedy by užívateľ nečakal. Pri porovnávaní postupu algoritmu so zahraničnými zdrojmi v prácach [3] [7] sa však tento postup zdá správny. Rozhodol som sa teda toto správanie algoritmu ponechať nezmenené. Na nasledujúcom obrázku je zreprodukovaný takýto prípad.

- 1. Koncový bod
- 2. Najbližší bod
- 3. Spojenie s týmto bodom by viedlo k nájdeniu cesty



Obr. 5.1: Bod Q_{goal} sa nespojí s grafom a algoritmus PRM ohlási nenájdenú cestu.[5]

Chyba v pseudokóde

Pseudokód na riadku 24 obsahuje logickú chybu, do grafu by sa mala pridať hrana $(goal, q^{'})$

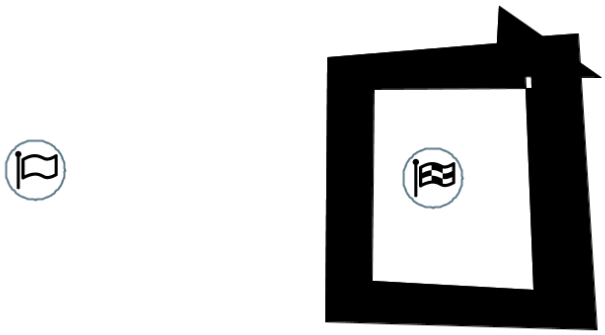
```
22.repeat
23.   if edge (goal,q') is not in obstacle
24.     add (start,q') to graph
25.   else
```

Obr. 5.2: Pravdepodobne chyba pri kopírovaní časti textu pseudokódu[5]

EST

Optimalizačná otázka

Pri izolovaní koncového bodu, ako na nasledujúcom obrázku, nemá algoritmus implementovaný žiadny limit maximálneho počtu vygenerovaných bodov. Podarilo sa mi teda aplikáciu „rozbiť“ keď som ju nechal bežať dostatočne dlhú dobu nad priestorom v ktorom neexistovala žiadna cesta medzi počiatočným a koncovým bodom. Aj keď toto nie je logická chyba, keďže samotný algoritmus nemá žiadne prostriedky na odhalenie takéhoto stavu, pre študentov môže byť tento stav nežiadúci. Možná oprava je napríklad implementácia maximálneho počtu vygenerovaných bodov/obmedzenie výpočetného času.



Obr. 5.3: Cieľ je nedostupný, aplikácia s algoritmom EST sa však donekonečna bude pokúšať zostaviť cestu.[5]

Do konfigurátora algoritmu bol pridaný vstupný parameter pre maximálny počet expanzií stromu. Toto pridanie zlepšilo zobrazovanie pridávania nových konfigurácií v prípade nedostupnosti cieľa.

Graf viditeľnosti

Nedostupné stavy

Tento algoritmus pri momentálnej implementácii nezvláda situáciu kedy je stav nedostupný. Užívateľ nedostane nijakú informáciu o tomto stave a algoritmus sa nespráva tak ako pri grafe s dosiahnuteľným cieľovým bodom. Už pri prvom kroku sa dostane aplikácia do stavu, kedy žiadne ďalšie kroky nevytvoria žiadnu zmenu. Do konzoly nie je vypísaný žiadny výstup a nezvýrazňujú sa žiadne riadky pseudo algoritmu. Aplikácia však do tohto stavu neprejde plynule počas prehľadávania, ale „skokom“ pri prvom kroku. V zdrojovom kóde algoritmu bolo teda nutné pridať kontrolu stavu nedostupnosti cieľovej konfigurácie a nastaviť zobrazenie tejto skutočnosti užívateľovi a korektne ukončiť výpočet algoritmu.

5.1.2 Algoritmy prehľadávania grafu

Nasledujúce algoritmy boli implementované Jakubom Rusnákom pri tvorbe jeho diplomovej práce [5].

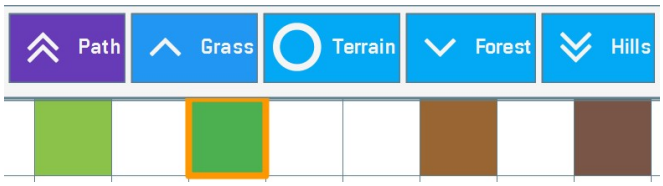
Bfs

Skrátený popis: Metóda prehľadávania do šírky (Breadth first search) je slepá metóda expandujúca stavový priestor do šírky rovnomerne. Expanzia začína v štarte a do fronty `OPEN` sa postupne pridávajú všetky susedné bunky tohoto stavu. Pre každý stav z fronty sa potom aplikuje rovnaký postup a takto sa postupne do fronty pridá celý stavový priestor ktorý sa prehľadáva kým nie sú prehľadané všetky bunky a už nemôžeme pridať žiadne nové alebo sa nenájde cieľová bunka a teda nájde cesta od štartu k cieľu. [8]

Tento algoritmus je z implementovaných najjednoduchší a pri výučbe umelých inteligencií sa používa ako základný príklad prehľadávania stavového priestoru práve kvôli jeho jednoduchosti. Rovnako nevyužíva hodnoty dĺžky jednotlivých hrán spájajúcich bunky a teda je kontrolovateľný pozorovaním vizualizácie. Pre každú prednastavenú mapu bol teda algoritmus krokovaný a správal sa podľa očakávaní.

Dijkstra

Skrátený popis: Dijkstrov algoritmus je neinformovaná metóda prehľadávania grafu. Pri vytváraní cesty grafom sa zaujíma o dĺžku trasy formou ohodnotených hrán grafu kde číselné ohodnotenie predstavuje dĺžku daného úseku. Rovnako ako aj ostatné algoritmy z tejto kategórie využíva prioritnú fronty `OPEN` do ktorej ukladá nájdené uzly na preskúmanie. V druhej množine `CLOSED` sa vytvára postupnosť prechádzania uzlov pre nájdenie najkratšej cesty grafom. V prípade, že sa po pridaní uzlu do množiny `CLOSED` nájde nový uzol s lepším ohodnotením, ten horší sa vymaže a nahradí novým. Dĺžka jednotlivých hrán sa simuluje farbou bunky ako je znázornené na obrázku 5.4 so stúpajúcou hodnotou zľava doprava.



Obr. 5.4: Nastavenie hodnoty dĺžky jednotlivej bunky pomocou farebných prepínačov. Dĺžka stúpa zľava doprava. [5]

Algoritmus má len 3 prednastavené mapy. Jeden s rovnomerným ohodnotením vzdialeností medzi bunkami a dva s rozdielnym ohodnoteným a simuláciou rozdielnej vzdialenosti buniek. Tieto bunky som pretvoril do grafu a simuloval nájdenie cesty pomocou nástroja [1]. S ručnou kontrolou množín `OPEN` a `CLOSED`.

Astar

Skrátený popis: Astar, alebo A* je algoritmus typu BestFS, ktorý využíva heuristickú funkciu $h(S_k)$ ako spodný odhad skutočnej cesty $h^*(S_k)$ od uzlu ktorý práve hodnotí k cieľu - táto heuristika sa potom nazýva prípustnou heuristikou [8].

Algoritmus A* využíva polohu cieľa na odhad heuristiky, jedná sa teda o informovanú metódu. Keďže algoritmus berie do úvahy dĺžky hrán medzi stavmi (bunkami) v aplikácií sa táto skutočnosť opäť simuluje farbami buniek ako na obrázku 5.4.

Pre testovanie algoritmu bol vyžitý rovnaký postup ako pri algoritme Dijkstra 5.1.2 a teda bol vytvorený príslušný graf prednastavenej mapy a porovnaný výsledok s nástrojom [1].

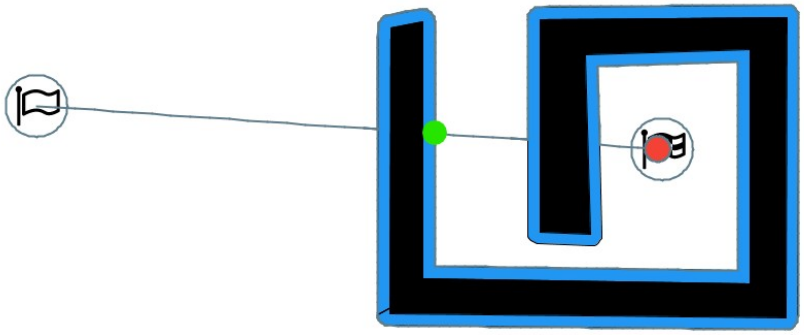
5.1.3 Bug algoritmy

Mnou vytvorené BUG algoritmy majú veľké rozlíšenie konfiguračného priestoru keďže sa nepohybujú po bunkách ale po súradniciach v 2D priestore. Nie je teda možné vypočítať presnú trasu robota grafom a porovnať ju s výslednou cestou simulácie algoritmu. Verifikácia a testovanie teda prebiehali viacerými spôsobmi.

- Po naštudovaní a pochopení algoritmu som pri spúšťaní jednotlivých presetov bug algoritmov hľadal nelogické správanie algoritmu.
- Kontrolné výpisy použitých premenných v určitých stavoch algoritmu. (napríklad uhol nájdenia prekážky, nameraná vzdialenosť obchádzania)

Bug a Bug II

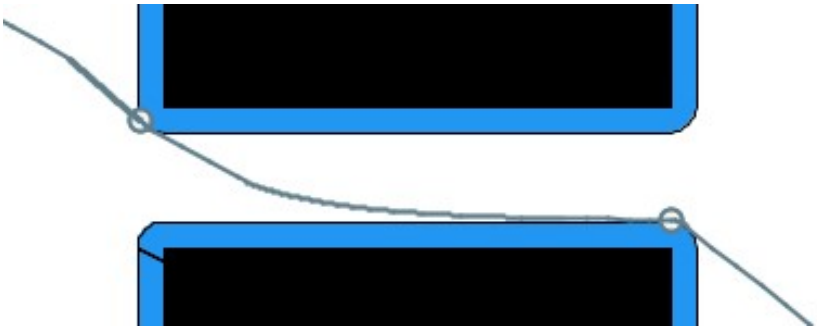
Algoritmy Bug a Bug II sú principiálne jednoduché. Ich celý popis je v sekcii 2.2.1 a 2.2.2, v základe však ide len o pohyb smerom k cieľu a obchádzanie prekážky jednou či druhou stranou podľa výberu užívateľa. Testoval som teda hlavne prepínač smeru obchádzania prekážky z rôznych kombinácií uhlov počiatočnej a koncovej konfigurácie.
Bug špecifické: Taktiež pri algoritme Bug bolo potrebné testovať zaznamenávanie prejdenej vzdialenosti na určenie smeru vrátenia sa k určitému bodu prekážky. Toto testovanie prebiehalo kombináciou vypisovania jednotlivých hodnôt a vizuálnym porovnaním smeru obchádzania.
Bug II špecifické: Pri algoritme Bug II bolo zase potrebné otestovať aj situáciu kedy robot narazí na prekážku netradičného tvaru a cez jeden „oddeľovaní“ bod kedy ukončuje obchádzanie prekážky musí prejsť viac ako raz 5.5. Toto testovanie prebiehalo vizuálnou verifikáciou, že sa robot nezacyklí.



Obr. 5.5: Prekážka netradičného tvaru a robot musí prejsť cez bod opustenia prekážky viac ako raz no v druhom a každom ďalšom cykle tento bod ignorovať. (Bod označený zelenou farbou) [5]

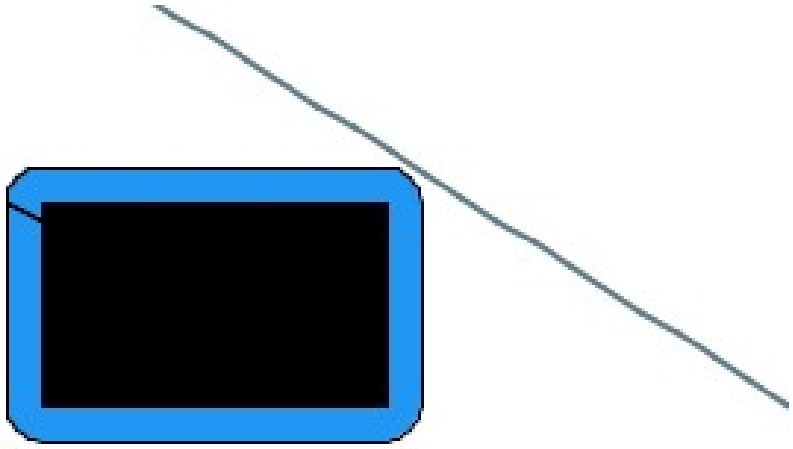
Tangent Bug

Algoritmus Tangent Bug je principiálne aj implementačne zložitejší ako algoritmy Bug a Bug II. Najjednoduchšou testovacou metódou bola vizuálna verifikácia kedy sa po vykreslení cesty algoritmom dá zhodnotiť optimálnosť nájdenej cesty a pristupovanie k obchádzaniu prekážok 5.6.



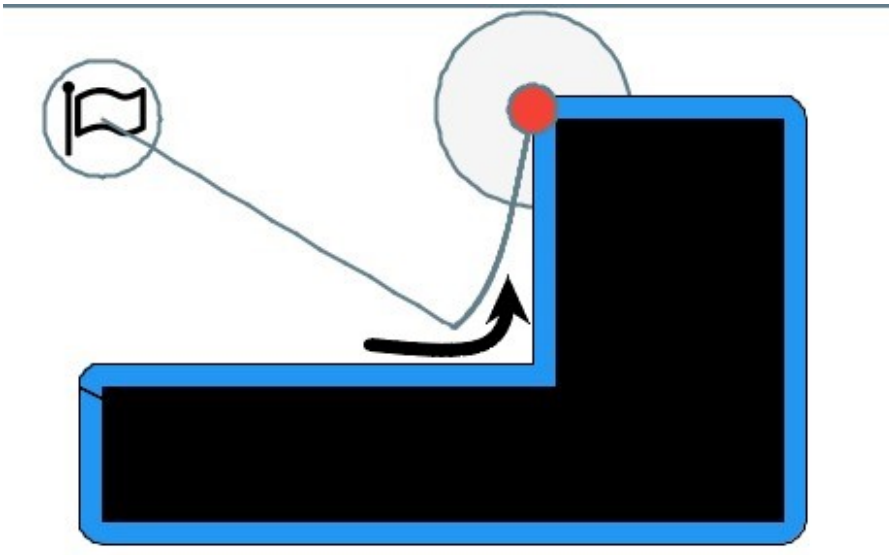
Obr. 5.6: Optimalizácia obchádzania prekážky algoritmom Tangent Bug - vizuálna verifikácia a pridanie bodov používaných algoritmom pri navigácii.

Algoritmus by mal zaznamenávať a reagovať len na tie prekážky, ktoré mu stoja v ceste, treba preto otestovať prípad, kedy robot prechádza tesne popri prekážke, no nenarazí na ňu 5.7.



Obr. 5.7: Prekážka je v blízkosti robota a v dosahu jeho senzorov, no nestojí mu v ceste a teda ju robot ignoruje.

Pri nájdení lokálneho minima minima počas obchádzania prekážky musí robot pokračovať k bodom, ktoré síce neznižujú heuristiku vzdialenosti k cieľu, no dovolia mu obísť prekážku. Takéto obchádzanie musí pokračovať v tom smere aký bol posledný pohyb robota k prekážke pred nájdením lokálneho minima aby bol tento pohyb optimálny 5.8.



Obr. 5.8: Robot nájde lokálne minimum prekážky a vyberie smer obchádzania prekážky.

Všetky ostatné situácie boli testované pozorovaním algoritmu v jednotlivých krokoch simulácie a vypisovaním premenných použitých pri výpočtoch.

Kapitola 6

Záver

V tejto práci som sa venoval analýze a štúdiu algoritmov pre plánovanie cesty v priestore a ich následnej simulácií v aplikácii využívajúcej knižnicu **vizlib** vytvorenej Jakubom Rusnákom v jeho diplomovej práci, ktorá uľahčuje prácu s GUI a celkovo zjednodušuje vizualizáciu rôznych algoritmov a dáva im podobný štýl a ovládanie.

V teoretickej časti som sa venoval prevažne popisu nových Bug algoritmov ktoré som implementoval a podrobnejšej analýze pravdepodobnostných algoritmov už predom implementovaných. Na tieto algoritmy bol braný väčší dôraz práve pre možnosti využitia aplikácie na študijné účely, a to prevažne pre študentov nižších ročníkov so snahou vytvorenia vizuálneho spojenia textu a reálneho postupu algoritmu. Takisto som opísal architektúru aplikácie pre jednoduchšie programovanie nových algoritmov pre programátorov, ktorý by chceli aplikáciu rozšíriť.

V rámci praktickej časti práce som implementoval tri nové algoritmy a upravil drobné nedostatky či preklepy a chyby algoritmov už implementovaných algoritmov. Mimo bodov zadania som sa venoval novej štruktúre simulačnej aplikácie, kde som z pôvodného návrhu samostatne stojacej aplikácie pre každý algoritmus vytvoril aplikáciu jednu, s možnosťou prepínania algoritmov. Do tejto aplikácie som potom pridal všetky už implementované algoritmy a tri moje vlastné.

Ďalšie rozšírenie mnou vytvorenej aplikácie môže byť určite implementácia nových algoritmov, napríklad využitie potenciálových funkcií, ktorých teoretický základ som uviedol v texte tejto práce. Ďalšie vhodné rozšírenie je vytvorenie jednoduchšieho vkladania nových algoritmov, ktoré by sa vkladali ako plug-in bez nutnosti hlbšieho pochopenia architektúry aplikácie.

Literatúra

[1] *Dijkstra and Astar simulation tool*. Dostupné z: https://www-m9.ma.tum.de/graph-algorithms/spp-dijkstra/index_en.html.

[2] Atyabi, A. a Powers, D. Review of classical and heuristic-based navigation and path planning approaches. *International Journal of Advancements in Computing Technology (IJACT)*. Január 2013, zv. 5, s. 1–14.

[3] Choset, H. *Principles of robot motion : theory, algorithms, and implementation*. The MIT Press, 2005. ISBN 0-262-03327-5.

[4] G.D. Hager, Z. D. a Mocha, D. Robotic Motion Planning: Configuration Space. Dostupné z: https://www.cs.cmu.edu/~motionplanning/lecture/Chap3-Config-Space_howie.pdf.

[5] Rusnák, J. *Vizualizace algoritmů pro plánování cesty*. Brno, CZ, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.vutbr.cz/studenti/zav-prace?zp_id=106120.

[6] Stojanovic, V. *Streaming of High Resolution Aerial Photography Textures Using a Web-Based Client/Server Model on Mobile Devices*. Hamburg, NL, 2016. Dizertačná práca. HafenCity University Hamburg.

[7] Švestka, P. a Overmars, M. H. Probabilistic path planning. In: Laumond, J. P., ed. *Robot Motion Planning and Control*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, s. 255–304. DOI: 10.1007/BFb0036074. ISBN 978-3-540-40917-5. Dostupné z: https://doi.org/10.1007/BFb0036074.

[8] Zbořil, F. V. 2021. Dostupné z: https://www.fit.vutbr.cz/study/courses/IZU/private/2021-opora-IZU.pdf.

Prílohy

Zoznam príloh

Príloha A

Obsah pamäťového média

- `text prace` - obsahuje pdf s textom práce a latex zdrojové súbory
- `Projekt` - Zdrojové kódy mnou vytvorenej aplikácie.
- `vizlib` - obsahuje knižnicu navrhnutú Jakubom Rusnákom potrebnú pre preloženie projektu. V podriečinku `src` sú zdrojové súbory, v `doc` sú UML návrhové diagramy pre program Visio od Microsoftu, v zložke `target` je skompilovaná java knižnica `vizlib` spolu s dokumentáciou a zdrojovými súbormy.
- `cz` - obsahuje knižnicu `vizlib`. Priečinok je určený pre skopírovanie do lokálneho repozitára `Mavenu` (`C:/Users/user/.m2`).
- `Template` - šablóna pre vytvorenie nového algoritmu s použitím knižnice `vizlib`.
- `Algorithm_visualisation.jar` - spustiteľný **.jar** súbor s mnou vytvorenou aplikácie.
- `Readme` - stručný návod na ovládanie aplikácie.

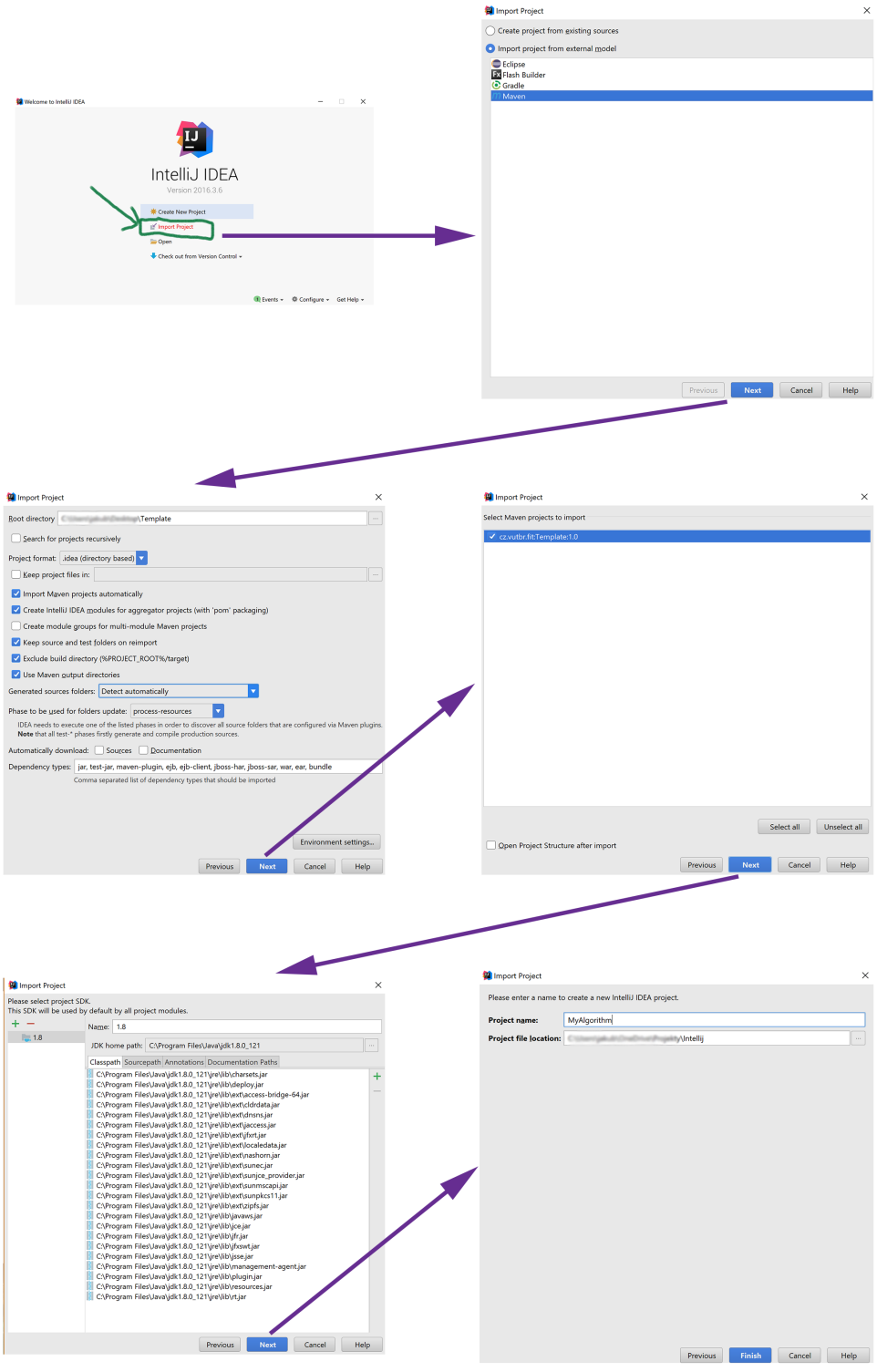
Príloha B

Vytvorenie projektu v IntelliJ a preloženie aplikácie

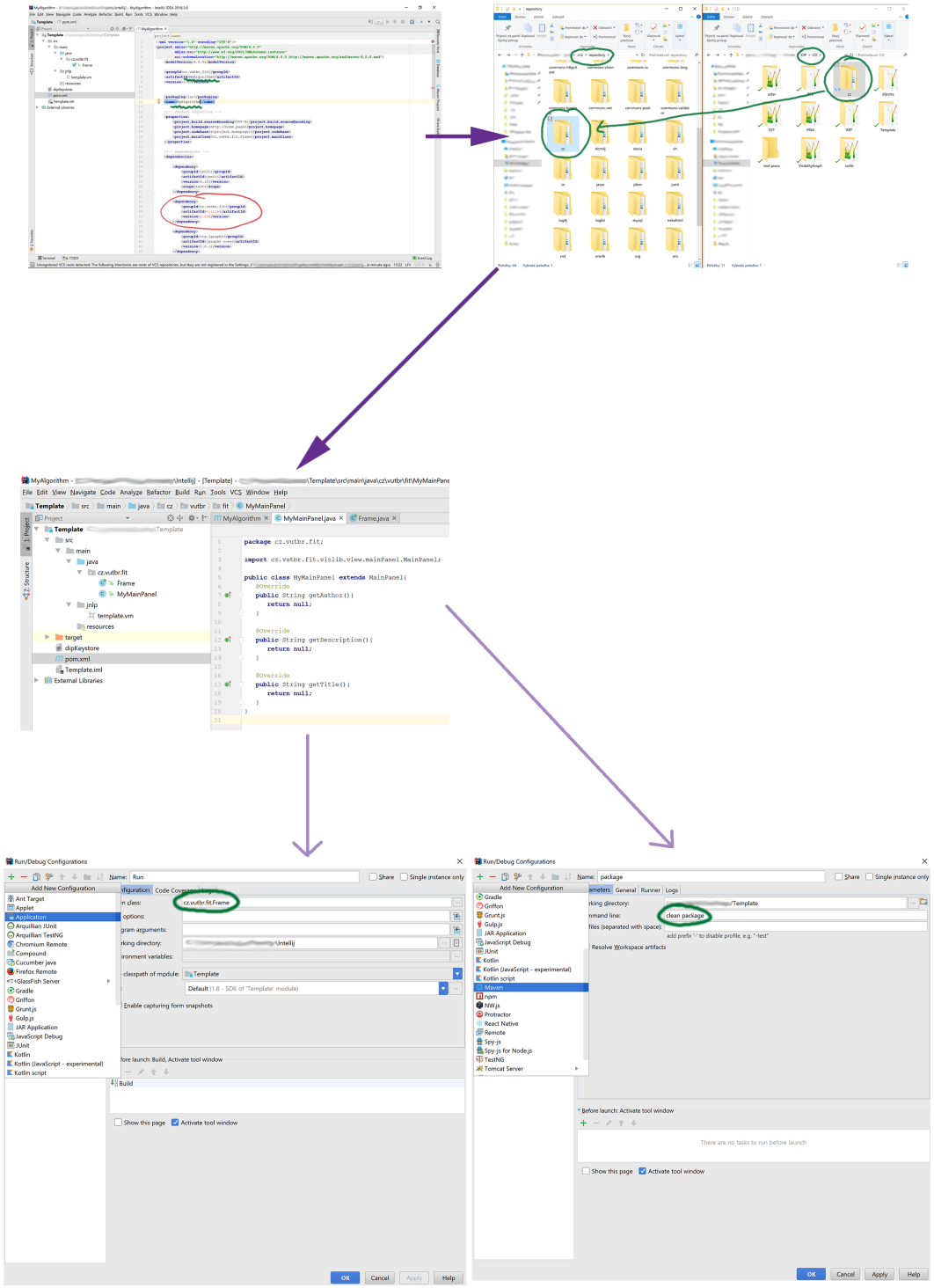
Jakub Rusnák vo svojej práci detailne popísal vytvorenie nového projektu v prostrediach InteliJ a Eclipse [5]. Pre úplnosť však poskytujem návod na vytvorenie a preloženie projektu v prostredí InteliJ, keďže som v ňom praktickú časť programoval. Postup je prevzatý z práce Jakuba Rusnáka [5] a upravený do aktuálnej podoby, sám som tento postup využil pri prvom spustení aplikácie a začatí úprav.

1. Nakopírujte priečinok `Projekt` z SD karty do pracovného adresára IntelliJ.
2. Pri spustení IntelliJ vyberte voľbu `Import Project` (obrázok B.1 prvá obrazovka).
3. Zvoľte `Import project from external model` a položku `Maven` (obrázok B.1 druhá obrazovka).
4. Vyberte `Projekt` ako `Root directory` (obrázok B.1 obrazovka 3).
5. Importujte nájdený projekt (obrázok B.1 obrazovka 4).
6. Zvoľte požadovanú SDK (obrázok B.1 obrazovka 5).
7. Pomenujte projekt (obrázok B.1 obrazovka 6).
8. Po vytvorení projektu sa zobrazí chyba (obrázok B.2 obrazovka 1), že systém nevedel nájsť knižnicu pre vizualizáciu algoritmov `vizlib` (pretože nie je v žiadnom repozitári). Vlastné knižnice sa musia nainštalovať do lokálneho repozitára `Maven`u, ktorý je zvyčajne v `C:/Users/user/.m2` priečinku. Inštalácia prebieha príkazom `mvn install` (zdroj https://maven.apache.org/guides/mini/guide-3rd-party-jars-local.html). Alternatívou je nakopírovanie potrebných súborov ako ukazuje obrázok B.2 obrazovka 2. Priečinok `cz`, ktorý obsahuje `vizlib` knižnicu sa nachádza na SD karte.
9. Nastavenie spustenia projektu ako java aplikácia je znázornené na obrázku B.2 obrazovka 4 - dole vľavo. Po zvolení konfigurácie `Application` sa nastaví hlavná trieda na rámec `Frame`, ktorý obsahuje vytvorený hlavný panel.
10. `Maven` konfigurácia je zobrazená na obrázku B.2 obrazovka 5 - dole vpravo. Po zvolení konfigurácie `Maven` sa nastavia argumenty príkazovej riadky na `clean package`. Tieto príkazy najprv odstránia všetky preložené súbory a potom zabalia celú aplikáciu do

jar archívov, vygenerujú javadoc a zdrojové súbory. Výsledná zabalená aplikácia sa nachádza v adresári projektu v priečinku `target`.



Obr. B.1: Vytvorenie spustiteľného projektu s aplikáciou v IntelliJ. Prevzaté z [5]

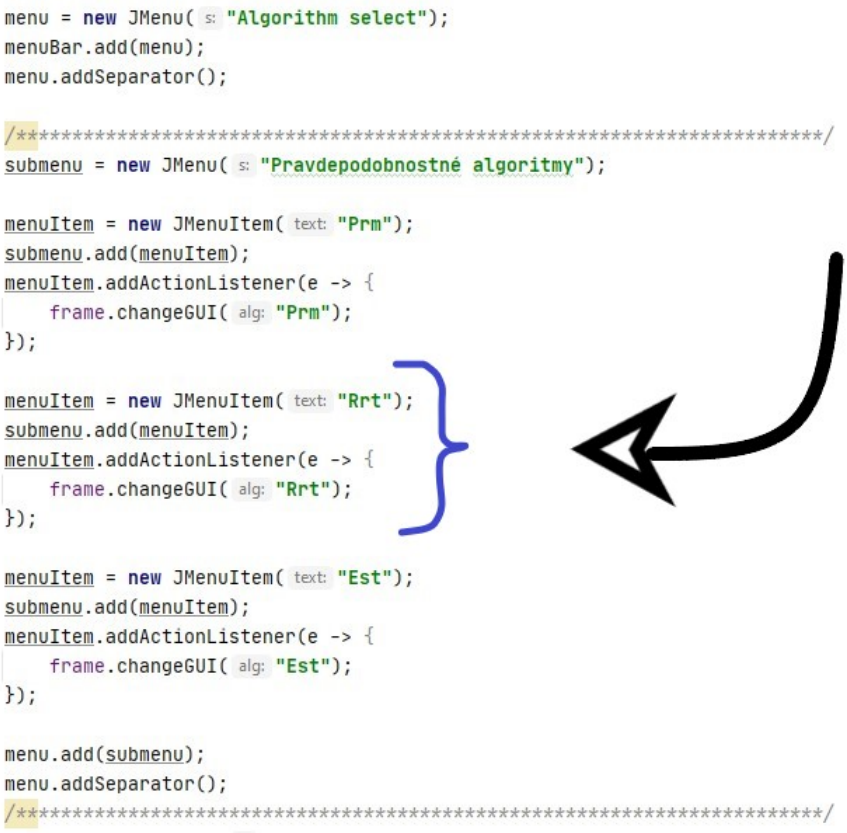


Obr. B.2: Vytvorenie spustiteľnej aplikácie v IntelliJ. Prevzaté z [5]

Príloha C

Pridanie nového algoritmu do aplikácie

1. V priečinku Template sa nachádzajú súbory pre uľahčenie tvorby nového algoritmu
2. V zložke **Template/emptyAlgorithms** sa nachádzajú pripravené .java súbory pohľadov na vytvorenie algoritmu. Zložku **Template/newAlgorithmViews** môžete premenovať a nakopírovať do adresára **src/main/java/cz/vutbr/fit/views**, v tomto priečinku budú pohľady nového algoritmu.
3. Vytvorte hlavný panel nového algoritmu, ktorý dedí od hlavného panela z knižnice. Najjednoduchší hlavný panel je na obrázku B.2 obrazovka 3 `MyMainPanel`.
4. Pridajte tento nový panel do prepínača menu **dropMenu** v zložke **src/main/java-/cz/vutbr/fit** ako na obrázku C.1, prípadne pridajte celé nové podmenu.
5. Pridajte nový hlavný panel a kartu do funkcie **changeAlg()** v súbore **Frame.java** ako na obrázku C.2



Obr. C.1: Pridanie algoritmu do menu dropMenu

```
JPanel newAlgCard = new JPanel();
cards.add(newAlgCard,  constraints: "AlgAcronym");

switch (alg){
   case "AlgAcronym":
      panel = new newAlgPanel( frame: this);
      frame.getContentPane().add(panel);
      break;
```

Obr. C.2: Pridanie nového algoritmu do prepínača algoritmov v súbore Frame.java